Developer Guide

AWS IoT Core

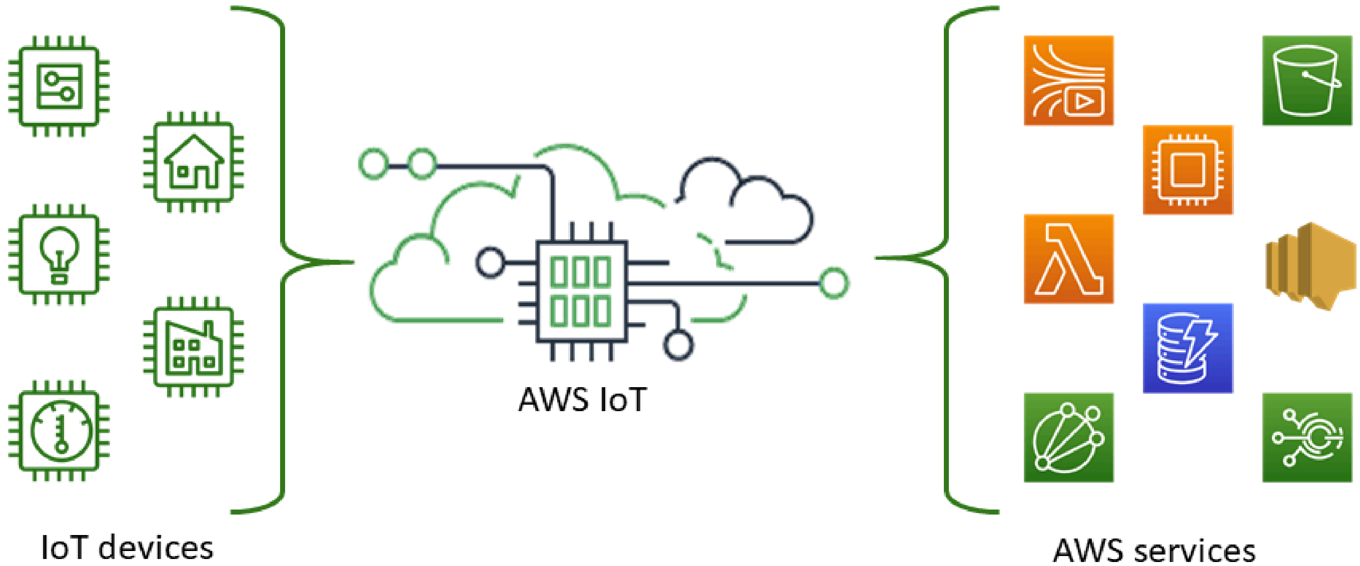
AWS IoT Core: Developer Guide

Table of Contents

What is AWS IoT?

AWS IoT provides the cloud services that connect your IoT devices to other devices and AWS cloud services. AWS IoT provides device software that can help you integrate your IoT devices into AWS IoT-based solutions. If your devices can connect to AWS IoT, AWS IoT can connect them to the cloud services that AWS provides.

For a hands-on introduction to AWS IoT, visit *Get started tutorials*.



With AWS IoT, you can select the most appropriate and up-to-date technologies for your solution. To help you manage and support your IoT devices in the field, AWS IoT Core supports these protocols:

- MQTT (Message Queuing and Telemetry Transport)
- MQTT over WSS (Websockets Secure)
- HTTPS (Hypertext Transfer Protocol - Secure)
- LoRaWAN (Long Range Wide Area Network)

The AWS IoT Core message broker supports devices and clients that use MQTT and MQTT over WSS protocols to publish and subscribe to messages. It also supports devices and clients that use the HTTPS protocol to publish messages.

AWS IoT Core for LoRaWAN helps you connect and manage wireless LoRaWAN (low-power long-range Wide Area Network) devices. AWS IoT Core for LoRaWAN replaces the need for you to develop and operate a LoRaWAN Network Server (LNS).

If you don't require AWS IoT features such as device communications, rules, or jobs, see AWS Messaging for information about other AWS IoT messaging services that might better fit your requirements.

How your devices and apps access AWS IoT

AWS IoT provides the following interfaces for AWS IoT tutorials:

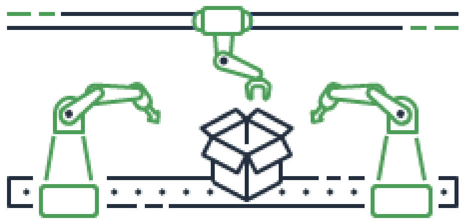
- **AWS IoT Device SDKs**—Build applications on your devices that send messages to and receive messages from AWS IoT. For more information, see *AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client*.
- **AWS IoT Core for LoRaWAN**—Connect and manage your long range WAN (LoRaWAN) devices and gateways by using AWS IoT Core for LoRaWAN.
- **AWS Command Line Interface (AWS CLI)**—Run commands for AWS IoT on Windows, macOS, and Linux. These commands allow you to create and manage thing objects, certificates, rules, jobs, and policies. To get started, see the AWS Command Line Interface User Guide. For more information about the commands for AWS IoT, see iot in the *AWS CLI Command Reference*.
- **AWS IoT API**—Build your IoT applications using HTTP or HTTPS requests. These API actions allow you to programmatically create and manage thing objects, certificates, rules, and policies. For more information about the API actions for AWS IoT, see Actions in the *AWS IoT API Reference*.
- **AWS SDKs**—Build your IoT applications using language-specific APIs. These SDKs wrap the HTTP/HTTPS API and allow you to program in any of the supported languages. For more information, see AWS SDKs and Tools.

You can also access AWS IoT through the AWS IoT console, which provides a graphical user interface (GUI) through which you can configure and manage the thing objects, certificates, rules, jobs, policies, and other elements of your IoT solutions.

What AWS IoT can do

This topic describes some of the solutions that you might need that AWS IoT supports.

IoT in Industry



These are some examples of AWS IoT solutions for industrial use cases that apply IoT technologies to improve the performance and productivity of industrial processes.

Solutions for industrial use cases

- **Use AWS IoT to build predictive quality models in industrial operations**

  See how AWS IoT can collect and analyze data from industrial operations to build predictive quality models. Learn more

- **Use AWS IoT to support predictive maintenance in industrial operations**

  See how AWS IoT can help plan preventive maintenance to reduce unplanned downtime. Learn more

IoT in Home automation

These are some examples of AWS IoT solutions for home automation use cases that apply IoT technologies to build scalable IoT applications that automate household activities using connected home devices.

Solutions for home automation

- **Use AWS IoT in your connected home**

See how AWS IoT can provide integrated home automation solutions.

- **Use AWS IoT to provide home security and monitoring**

  See how AWS IoT can apply machine learning and edge computing to your home automation solution.

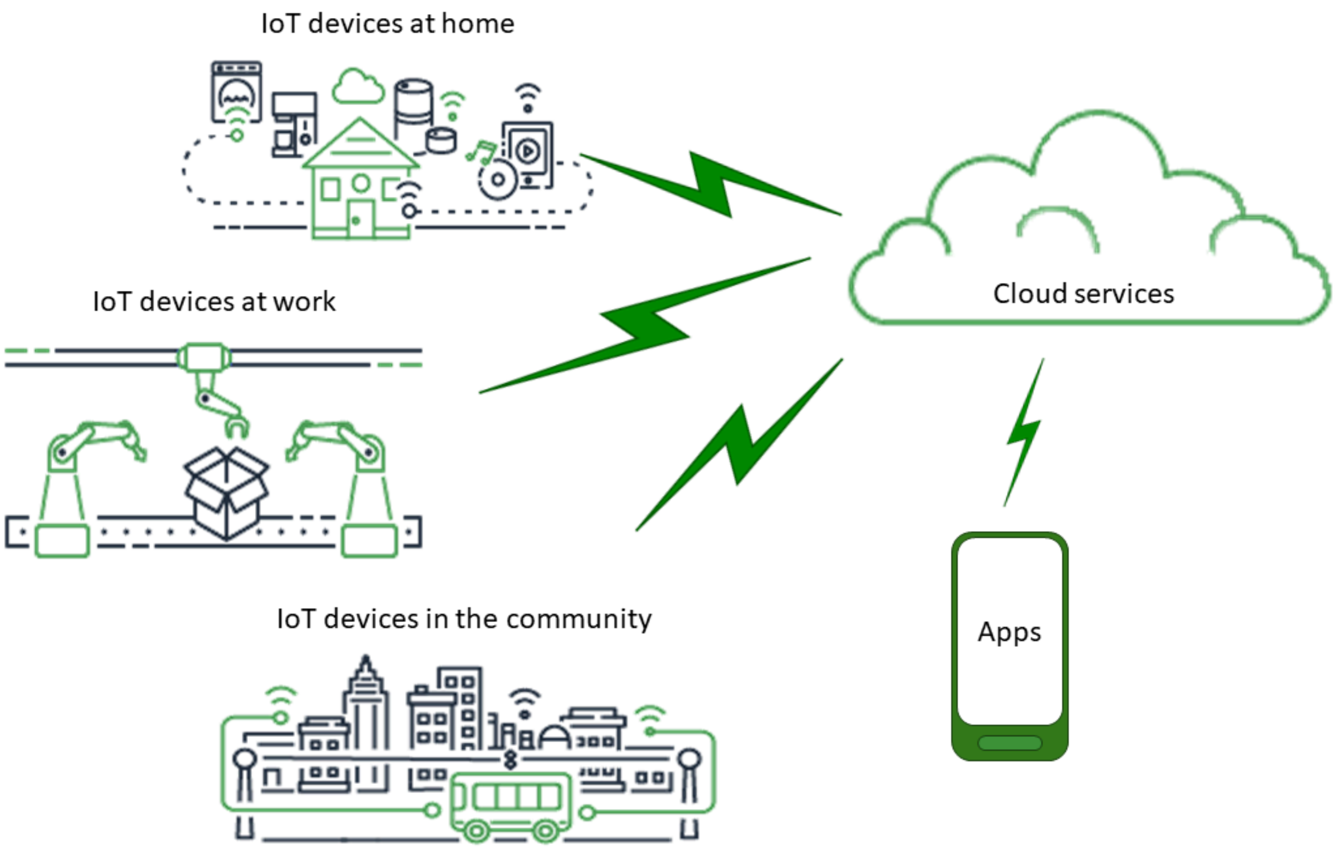
For a list of solutions for industrial, consumer, and commercial use cases, see the AWS IoT Solution Repository.

How AWS IoT works

AWS IoT provides cloud services and device support that you can use to implement IoT solutions. AWS provides many cloud services to support IoT-based applications. So to help you understand where to start, this section provides a diagram and definition of essential concepts to introduce you to the IoT universe.

The IoT universe

In general, the Internet of Things (IoT) consists of the key components shown in this diagram.



Apps

Apps give end users access to IoT devices and the features provided by the cloud services to which those devices are connected.

Cloud services

Cloud services are distributed, large-scale data storage and processing services that are connected to the internet. Examples include:

- IoT connection and management services

  AWS IoT is an example of an IoT connection and management service.
- Compute services, such as Amazon Elastic Compute Cloud and AWS Lambda
- Database services, such as Amazon DynamoDB

Communications

Devices communicate with cloud services by using various technologies and protocols. Examples include:

- Wi-Fi/Broadband internet
- Broadband cellular data
- Narrow-band cellular data
- Long-range Wide Area Network (LoRaWAN)
- Proprietary RF communications

Devices

A device is a type of hardware that manages interfaces and communications. Devices are usually located in close proximity to the real-world interfaces they monitor and control. Devices can include computing and storage resources, such as microcontrollers, CPU, memory. Examples include:

- Raspberry Pi
- Arduino
- Voice-interface assistants
- LoRaWAN and devices
- Amazon Sidewalk devices
- Custom IoT devices

Interfaces

An interface is a component that connects a device to the physical world.

- User interfaces

  Components that allow devices and users to communicate with each other.

  - Input interfaces

    Enable a user to communicate with a device

Examples: keypad, button

- Output interfaces

  Enable a device to communicate with a user

  Examples: Alpha-numeric display, graphical display, indicator light, alarm bell

- Sensors

  Input components that measure or sense something in the outside world in a way that a device understands. Examples include:

  - Temperature sensor (converts temperature to an analog or digital signal)
  - Humidity sensor (converts relative humidity to an analog or digital signal)
  - Analog to digital convertor (converts an analog voltage to a numeric value)
  - Ultrasonic distance measuring unit (converts a distance to a numeric value)
  - Optical sensor (converts a light level to a numeric value)
  - Camera (converts image data to digital data)

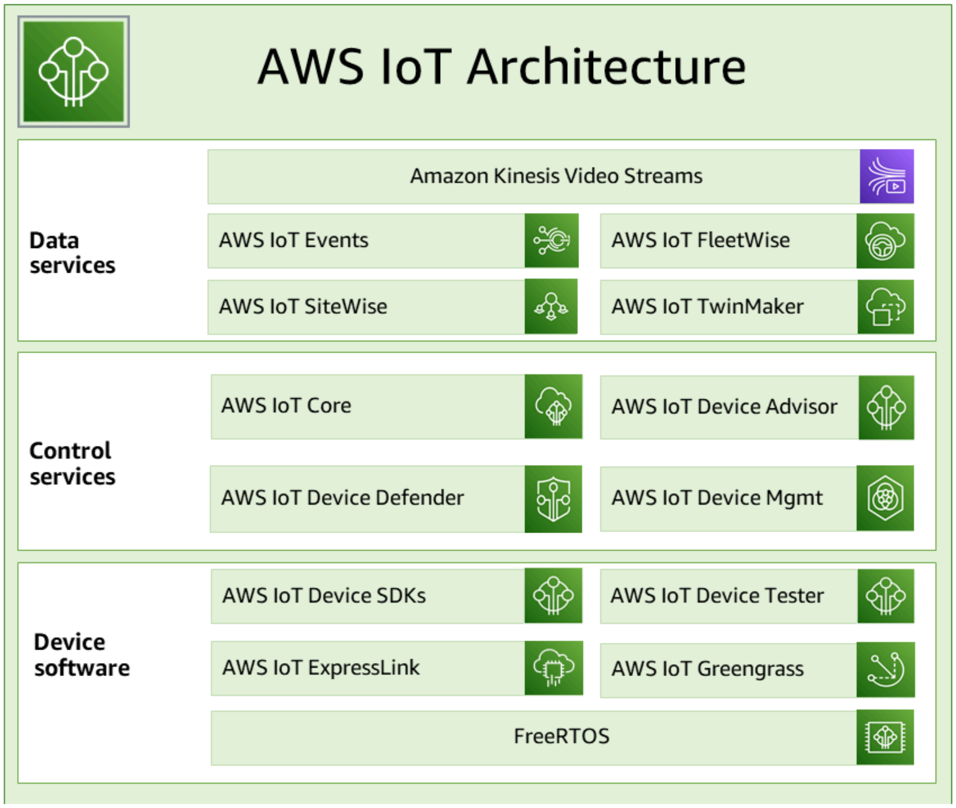
- Actuators

  Output components that the device can use to control something in the outside world. Examples include:

  - Stepper motors (convert electric signals to movement)
  - Relays (control high electric voltages and currents)

AWS IoT services overview

In the IoT universe, AWS IoT provides the services that support the devices that interact with the world and the data that passes between them and AWS IoT. AWS IoT is made up of the services that are shown in this illustration to support your IoT solution.



AWS IoT device software

AWS IoT provides this software to support your IoT devices.

AWS IoT Device SDKs

The AWS IoT Device and Mobile SDKs help you efficiently connect your devices to AWS IoT. The AWS IoT Device and Mobile SDKs include open-source libraries, developer guides with samples, and porting guides so that you can build innovative IoT products or solutions on your choice of hardware platforms.

AWS IoT Device Tester

AWS IoT Device Tester for FreeRTOS and AWS IoT Greengrass is a test automation tool for microcontrollers. AWS IoT Device Tester tests your device to determine if it will run FreeRTOS or AWS IoT Greengrass and interoperate with AWS IoT services.

AWS IoT ExpressLink

AWS IoT ExpressLink powers a range of hardware modules developed and offered by AWS Partners. The connectivity modules include AWS-validated software, making it faster and easier for you to securely connect devices to the cloud and seamlessly integrate with a range of AWS services. For more information, visit the AWS IoT ExpressLink overview page or see the AWS IoT ExpressLink Programmer's Guide.

AWS IoT Greengrass

AWS IoT Greengrass extends AWS IoT to edge devices so they can act locally on the data they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables your devices to collect and analyze data closer to where that data is generated, react autonomously to local events, and communicate securely with other devices on the local network. You can use AWS IoT Greengrass to build edge applications using pre-built software modules, called components, that can connect your edge devices to AWS services or third-party services.

FreeRTOS

FreeRTOS is an open source, real-time operating system for microcontrollers that lets you include small, low-power edge devices in your IoT solution. FreeRTOS includes a kernel and a growing set of software libraries that support many applications. FreeRTOS systems can securely connect your small, low-power devices to AWS IoT and support more powerful edge devices running AWS IoT Greengrass.

AWS IoT control services

Connect to the following AWS IoT services to manage the devices in your IoT solution.

AWS IoT Core

AWS IoT Core is a managed cloud service that enables connected devices to securely interact with cloud applications and other devices. AWS IoT Core can support many devices and messages, and it can process and route those messages to AWS IoT endpoints and other devices. With AWS IoT Core, your applications can interact with all of your devices even when they aren’t connected.

AWS IoT Core Device Advisor

AWS IoT Core Device Advisor is a cloud-based, fully managed test capability for validating IoT devices during device software development. Device Advisor provides pre-built tests that you can use to validate IoT devices for reliable and secure connectivity with AWS IoT Core, before deploying devices to production.

AWS IoT Device Defender

AWS IoT Device Defender helps you secure your fleet of IoT devices. AWS IoT Device Defender continuously audits your IoT configurations to make sure that they aren't deviating from security best practices. AWS IoT Device Defender sends an alert when it detects any gaps in your IoT configuration that might create a security risk, such as identity certificates being shared across multiple devices or a device with a revoked identity certificate trying to connect to AWS IoT Core.

AWS IoT Device Management

AWS IoT Device Management services help you track, monitor, and manage the plethora of connected devices that make up your device fleets. AWS IoT Device Management services help you ensure that your IoT devices work properly and securely after they have been deployed. They also provide secure tunneling to access your devices, monitor their health, detect and remotely troubleshoot problems, as well as services to manage device software and firmware updates.

AWS IoT data services

Analyze the data from the devices in your IoT solution and take appropriate action by using the following AWS IoT services.

Amazon Kinesis Video Streams

Amazon Kinesis Video Streams allows you to stream live video from devices to the AWS Cloud, where it is durably stored, encrypted, and indexed, allowing you to access your data through easy-to-use APIs. You can use Amazon Kinesis Video Streams to capture massive amounts of live video data from millions of sources, including smartphones, security cameras, webcams, cameras embedded in cars, drones, and other sources. Amazon Kinesis Video Streams enables you to play back video for live and on-demand viewing, and quickly build applications that take advantage of computer vision and video analytics through integration with Amazon Rekognition Video, and libraries for ML frameworks. You can also send non-video time-serialized data such as audio data, thermal imagery, depth data, RADAR data, and more.

Amazon Kinesis Video Streams with WebRTC

Amazon Kinesis Video Streams with WebRTC provides a standards-compliant WebRTC implementation as a fully managed capability. You can use Amazon Kinesis Video Streams with WebRTC to securely live stream media or perform two-way audio or video interaction between any camera IoT device and WebRTC-compliant mobile or web players. As a fully managed capability, you don't have to build, operate, or scale any WebRTC-related cloud infrastructure, such as signaling or media relay servers to securely stream media across applications and devices. Using Amazon Kinesis Video Streams with WebRTC, you can easily build applications for live peer-to-peer media streaming, or real-time audio or video interactivity between camera IoT devices, web browsers, and mobile devices for a variety of use cases.

AWS IoT Events

AWS IoT Events detects and responds to events from IoT sensors and applications. Events are patterns of data that identify more complicated circumstances than expected, such as motion detectors using movement signals to activate lights and security cameras. AWS IoT Events continuously monitors data from multiple IoT sensors and applications, and integrates with other services, such as AWS IoT Core, IoT SiteWise, DynamoDB, and others to enable early detection and unique insights.

AWS IoT FleetWise

AWS IoT FleetWise is a managed service that you can use to collect and transfer vehicle data to the cloud in near-real time. With AWS IoT FleetWise, you can easily collect and organize data from vehicles that use different protocols and data formats. AWS IoT FleetWise helps to transform low-level messages into human-readable values and standardize the data format in the cloud for data analyses. You can also define data collection schemes to control what data to collect in vehicles and when to transfer it to the cloud.

AWS IoT SiteWise

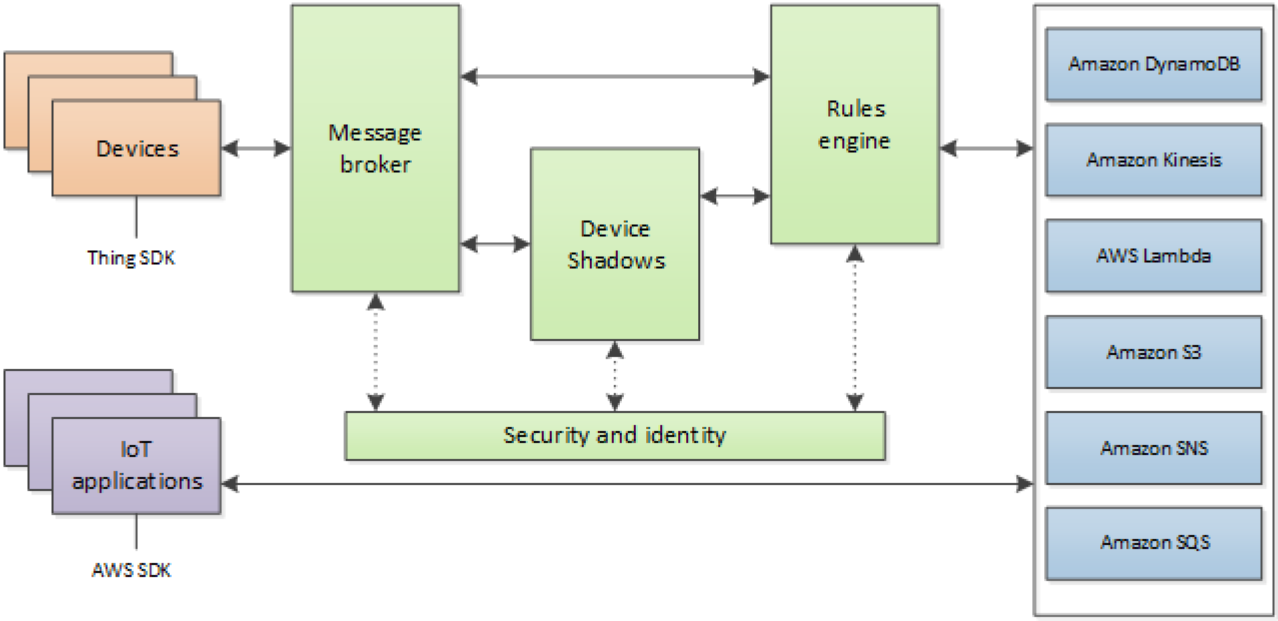
AWS IoT SiteWise collects, stores, organizes, and monitors data passed from industrial equipment by MQTT messages or APIs at scale by providing software that runs on a gateway in your facilities. The gateway securely connects to your on-premises data servers and automates the process of collecting and organizing the data and sending it to the AWS Cloud.

AWS IoT TwinMaker

AWS IoT TwinMaker builds operational digital twins of physical and digital systems. AWS IoT TwinMaker creates digital visualizations using measurements and analysis from a variety of real-world sensors, cameras, and enterprise applications to help you keep track of your physical factory, building, or industrial plant. You can use real-world data to monitor operations, diagnose and correct errors, and optimize operations.

AWS IoT Core services

AWS IoT Core provides the services that connect your IoT devices to the AWS Cloud so that other cloud services and applications can interact with your internet-connected devices.



The next section describes each of the AWS IoT Core services shown in the illustration.

AWS IoT Core messaging services

The AWS IoT Core connectivity services provide secure communication with the IoT devices and manage the messages that pass between them and AWS IoT.

Device gateway

Enables devices to securely and efficiently communicate with AWS IoT. Device communication is secured by secure protocols that use X.509 certificates.

Message broker

Provides a secure mechanism for devices and AWS IoT applications to publish and receive messages from each other. You can use either the MQTT protocol directly or MQTT over WebSocket to publish and subscribe. For more information about the protocols that AWS IoT supports, see the section called "Device communication protocols". Devices and clients can also use the HTTP REST interface to publish data to the message broker.

The message broker distributes device data to devices that have subscribed to it and to other AWS IoT Core services, such as the Device Shadow service and the rules engine.

AWS IoT Core for LoRaWAN

AWS IoT Core for LoRaWAN makes it possible to set up a private LoRaWAN network by connecting your LoRaWAN devices and gateways to AWS without the need to develop and operate a LoRaWAN Network Server (LNS). Messages received from LoRaWAN devices are sent to the rules engine where they can be formatted and sent to other AWS IoT services.

Rules engine

The Rules engine connects data from the message broker to other AWS IoT services for storage and additional processing. For example, you can insert, update, or query a DynamoDB table or invoke a Lambda function based on an expression that you defined in the Rules engine. You can use an SQL-based language to select data from message payloads, and then process and send the data to other services, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and AWS Lambda. You can also create rules that republish messages to the message broker and on to other subscribers. For more information, see Rules for AWS IoT.

AWS IoT Core control services

The AWS IoT Core control services provide device security, management, and registration features.

Custom Authentication service

You can define custom authorizers that allow you to manage your own authentication and authorization strategy using a custom authentication service and a Lambda function. Custom authorizers allow AWS IoT to authenticate your devices and authorize operations using bearer token authentication and authorization strategies.

Custom authorizers can implement various authentication strategies; for example, JSON Web Token verification or OAuth provider callout. They must return policy documents that are used by the device gateway to authorize MQTT operations. For more information, see Custom authentication and authorization.

Device Provisioning service

Allows you to provision devices using a template that describes the resources required for your device: a *thing object*, a certificate, and one or more policies. A thing object is an entry in the registry that contains attributes that describe a device. Devices use certificates to authenticate with AWS IoT. Policies determine which operations a device can perform in AWS IoT.

The templates contain variables that are replaced by values in a dictionary (map). You can use the same template to provision multiple devices just by passing in different values for the template variables in the dictionary. For more information, see Device provisioning.

Group registry

Groups allow you to manage several devices at once by categorizing them into groups. Groups can also contain groups—you can build a hierarchy of groups. Any action that you perform on a parent group will apply to its child groups. The same action also applies to all the devices in the parent group and all devices in the child groups. Permissions granted to a group will apply to all devices in the group and in all of its child groups. For more information, see Managing devices with AWS IoT.

Jobs service

Allows you to define a set of remote operations that are sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install application or firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

To create a job, you specify a description of the remote operations to be performed and a list of targets that should perform them. The targets can be individual devices, groups or both. For more information, see AWS IoT Jobs.

Registry

Organizes the resources associated with each device in the AWS Cloud. You register your devices and associate up to three custom attributes with each one. You can also associate certificates and MQTT client IDs with each device to improve your ability to manage and troubleshoot them. For more information, see Managing devices with AWS IoT.

Security and Identity service

Provides shared responsibility for security in the AWS Cloud. Your devices must keep their credentials safe to securely send data to the message broker. The message broker and rules engine use AWS security features to send data securely to devices or other AWS services. For more information, see Authentication.

AWS IoT Core data services

The AWS IoT Core data services help your IoT solutions provide a reliable application experience even with devices that are not always connected.

Device shadow

A JSON document used to store and retrieve current state information for a device.

Device Shadow service

The Device Shadow service maintains a device's state so that applications can communicate with a device whether the device is online or not. When a device is offline, the Device Shadow service manages its data for connected applications. When the device reconnects, it synchronizes its state with that of its shadow in the Device Shadow service. Your devices can also publish their current state to a shadow for use by applications or other devices that might not be connected all the time. For more information, see AWS IoT Device Shadow service.

AWS IoT Core support service

Amazon Sidewalk Integration for AWS IoT Core

Amazon Sidewalk is a shared network that improves connectivity options to help devices work together better. Amazon Sidewalk supports a wide range of customer devices such as those that locate pets or valuables, those that provide smart home security and lighting control, and those that provide remote diagnostics for appliances and tools. Amazon Sidewalk Integration for AWS IoT Core makes it possible for device manufacturers to add their Sidewalk device fleet to the AWS IoT Cloud.

For more information, see AWS IoT Core for Amazon Sidewalk.

Learn more about AWS IoT

This topic helps you get familiar with the world of AWS IoT. You can get general information about how IoT solutions are applied in various use cases, training resources, links to social media for AWS IoT and all other AWS services, and a list of services and communication protocols that AWS IoT uses.

Training resources for AWS IoT

We provide these training courses to help you learn about AWS IoT and how to apply them to your solution design.

- **Introduction to AWS IoT**

A video overview of AWS IoT and its core services.

- **Deep Dive into AWS IoT Authentication and Authorization**

  An advanced course that explores the concepts of AWS IoT authentication and authorization. You will learn how to authenticate and authorize clients to access the AWS IoT control plane and data plane APIs.

- **Internet of Things Foundation Series**

  A learning path of IoT eLearning modules on different IoT technologies and features.

AWS IoT resources and guides

These are in-depth technical resources on specific aspects of AWS IoT.

- **IoT Lens – AWS IoT Well-Architected Framework**

  A document that describes the best practices for architecting your IoT applications on AWS.

- **Designing MQTT Topics for AWS IoT Core**

  A whitepaper that describes the best practices for designing MQTT topics in AWS IoT Core and leveraging AWS IoT Core features with MQTT.

- **Abstract and introduction**

  A PDF document that describes the different ways that AWS IoT provides to provision large fleets of devices.

- **AWS IoT Core Device Advisor**

  AWS IoT Core Device Advisor provides pre-built tests that you can use to validate IoT devices for reliable and secure connectivity best practices with AWS IoT Core, before deploying devices to production.

- **AWS IoT Resources**

  IoT-specific resources, such as Technical Guides, Reference Architectures, eBooks, and curated blog posts, presented in a searchable index.

- **IoT Atlas**

Overviews on how to solve common IoT design problems. The *IoT Atlas* provides in-depth looks into the design challenges that you are likely to encounter while developing your IoT solution.

- **AWS Whitepapers & Guides**

  Our current collection of whitepapers and guides on AWS IoT and other AWS technologies.

AWS IoT in social media

These social media channels provide information about AWS IoT and AWS-related topics.

- The Internet of Things on AWS IoT – Official Blog
- AWS IoT videos in the Amazon Web Services channel on YouTube

These social media accounts cover all AWS services, including AWS IoT

- The Amazon Web Services channel on YouTube
- Amazon Web Services on Twitter
- Amazon Web Services on Facebook
- Amazon Web Services on Instagram
- Amazon Web Services on LinkedIn

AWS services used by the AWS IoT Core rules engine

The AWS IoT Core rules engine can connect to these AWS services.

- **Amazon DynamoDB**

  Amazon DynamoDB is a scalable, NoSQL database service that provides fast and predictable database performance.

- **Amazon Kinesis**

  Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data so you can get timely insights and react quickly to new information. Amazon Kinesis can ingest real-time data such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications.

- **AWS Lambda**

  AWS Lambda lets you run code without provisioning or managing servers. You can set up your code to automatically trigger from AWS IoT data and events or call it directly from a web or mobile app.
- **Amazon Simple Storage Service**

  Amazon Simple Storage Service (Amazon S3) can store and retrieve any amount of data at any time, from anywhere on the web. AWS IoT rules can send data to Amazon S3 for storage.
- **Amazon Simple Notification Service**

  Amazon Simple Notification Service (Amazon SNS) is a web service that enables applications, end users, and devices to send and receive notifications from the cloud.
- **Amazon Simple Queue Service**

  Amazon Simple Queue Service (Amazon SQS) is a message queuing service that decouples and scales microservices, distributed systems, and serverless applications.
- **Amazon OpenSearch Service**

  Amazon OpenSearch Service (OpenSearch Service) is a managed service that makes it easy to deploy, operate, and scale OpenSearch, a popular open-source search and analytics engine.
- **Amazon SageMaker AI**

  Amazon SageMaker AI can create machine learning (ML) models by finding patterns in your IoT data. The service uses these models to process new data and generate predictions for your application.
- **Amazon CloudWatch**

  Amazon CloudWatch provides a reliable, scalable, and flexible monitoring solution to help set up, manage, and scale your own monitoring systems and infrastructure.

Communication protocols supported by AWS IoT Core

These topics provide more information about the communication protocols used by AWS IoT. For more information about the protocols used by AWS IoT and connecting devices and services to AWS IoT, see Connect to AWS IoT Core.

- **MQTT (Message Queuing Telemetry Transport)**

The home page of the MQTT.org site where you can find the MQTT protocol specifications. For more information about how AWS IoT supports MQTT, see MQTT.

- **HTTPS (Hypertext Transfer Protocol - Secure)**

  Devices and apps can access AWS IoT services by using HTTPS.

- **LoRaWAN (Long Range Wide Area Network)**

  LoRaWAN devices and gateways can connect to AWS IoT Core by using AWS IoT Core for LoRaWAN.

- **TLS (Transport Layer Security) v1.3**

  The specification of the TLS v1.3 (RFC 5246). AWS IoT uses TLS v1.3 to establish secure connections between devices and AWS IoT.

What's new in the AWS IoT console

We're in the process of updating the user interface of the AWS IoT console to a new experience. We're updating the user interface in stages, so some pages in the console will have a new experience, some might have both the original and the new experience, and some might have only the original experience.

This table displays the state of individual areas of the AWS IoT console user interface as of January 27, 2022.

AWS IoT console user interface status

Console page	Original experience	New experience	Comments
Monitor	Not available	Available	
Activity	Not available	Available	
Onboard - Get started	Not available	Available	Not available in CN Regions
Onboard - Fleet provisioning templates	Available	Available	

Console page	Original experience	New experience	Comments
Manage - Things	Available	Available	
Manage - Types	Available	Available	
Manage - Thing groups	Available	Available	
Manage - Billing groups	Available	Available	
Manage - Jobs	Available	Available	
Manage - Job templates	Not available	Available	
Manage - Tunnels	Not available	Available	
Fleet Hub - Get started	Not available	Available	Not available in all AWS Regions
Fleet Hub - Applicati ons	Not available	Available	Not available in all AWS Regions
Greengrass - Getting started	Not available	Available	Not available in all AWS Regions
Greengrass - Core devices	Not available	Available	Not available in all AWS Regions
Greengrass - Components	Not available	Available	Not available in all AWS Regions
Greengrass - Deployments	Not available	Available	Not available in all AWS Regions
Greengrass - Classic (V1)	Available	Available	

Console page	Original experience	New experience	Comments
Wireless connectivity - Intro	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Gateways	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Devices	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Profiles	Not available	Available	Not available in all AWS Regions
Wireless connectivity - Destinations	Not available	Available	Not available in all AWS Regions
Secure - Certificates	Available	Available	
Secure - Policies	Available	Available	
Secure - CAs	Available	Available	
Secure - Role Aliases	Available	Available	
Secure - Authorizers	Available	Available	
Defend - Intro	Not available	Available	
Defend - Audit	Not available	Available	
Defend - Detect	Not available	Available	
Defend - Mitigation actions	Not available	Available	
Defend - Settings	Not available	Available	
Act - Rules	Available	Available	
Act - Destinations	Available	Available	

Console page	Original experience	New experience	Comments
Test - Device Advisor	Available	Available	Not available in all AWS Regions
Test - MQTT test client	Available	Available	
Software	Available	Available	
Settings	Not available	Available	
Learn	Available	Not available yet	

Legend

Status values

- **Available**

  This user interface experience can be used.
- **Not available**

  This user interface experience can't be used.
- **Not available yet**

  The new user interface experience is being worked on, but it's not ready, yet.
- **In progress**

  The new user interface experience is in the process of being updated. Some pages might still have the original user experience, however.

Using AWS IoT with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Getting started with AWS IoT Core tutorials

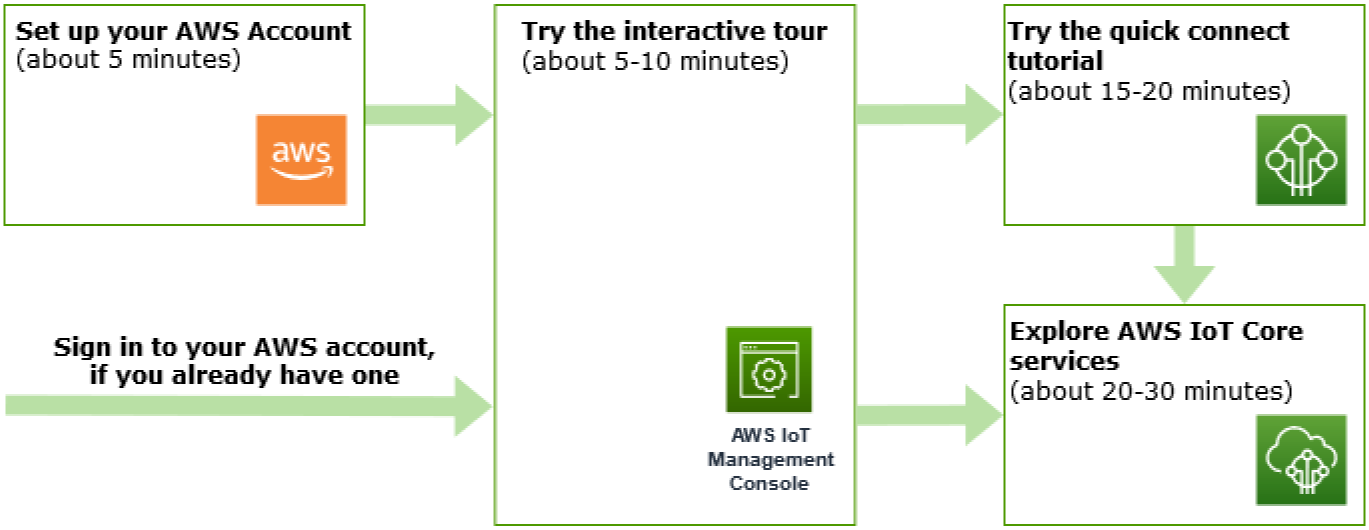
Whether you're new to IoT or you have years of experience, these resources present the AWS IoT concepts and terms that will help you start using AWS IoT.

- **Look** inside AWS IoT and its components in How AWS IoT works.
- **Learn** more about AWS IoT from our collection of training materials and videos. This topic also includes a list of services that AWS IoT can connect to, social media links, and links to communication protocol specifications.
- **the section called "Connect your first device to AWS IoT Core"**.
- **Develop** your IoT solutions by Connect to AWS IoT Core and exploring the AWS IoT tutorials.
- **Test and validate** your IoT devices for secure and reliable communication by using the Device Advisor.
- **Manage** your solution by using AWS IoT Core management services such as Fleet indexing, AWS IoT Jobs, and AWS IoT Device Defender.
- **Analyze** the data from your devices by using the AWS IoT data services.

Connect your first device to AWS IoT Core

AWS IoT Core services connect IoT devices to AWS IoT services and other AWS services. AWS IoT Core includes the device gateway and the message broker, which connect and process messages between your IoT devices and the cloud.

Here's how you can get started with AWS IoT Core and AWS IoT.



This section presents a tour of the AWS IoT Core to introduce its key services and provides several examples of how to connect a device to AWS IoT Core and pass messages between them. Passing messages between devices and the cloud is fundamental to every IoT solution and is how your devices can interact with other AWS services.

- **Set up AWS account**

  Before you can use AWS IoT services, you must set up an AWS account. If you already have an AWS account and an IAM user for yourself, you can use them and skip this step.
- **Try the quick connect tutorial**

  This tutorial is best if you want to quickly get started with AWS IoT and see how it works in a limited scenario. In this tutorial, you'll need a device and you'll install some AWS IoT software on it. If you don't have an IoT device, you can use your Windows, Linux, or macOS personal computer as a device for this tutorial. If you want to try AWS IoT, but you don't have a device, try the next option.
- **Try the interactive tutorial**

  This demo is best if you want to see what a basic AWS IoT solution can do without connecting a device or downloading any software. The interactive tutorial presents a simulated solution built on AWS IoT Core services that illustrates how they interact.
- **Explore AWS IoT Core services with a hands-on tutorial**

  This tutorial is best for developers who want to get started with AWS IoT so they can continue to explore other AWS IoT Core features such as the rules engine and shadows. This tutorial follows a process similar to the quick connect tutorial, but provides more details on each step to enable a smoother transition to the more advanced tutorials.
- **View MQTT messages with the AWS IoT MQTT client**

  Learn how to use the MQTT test client to watch your first device publish MQTT messages to AWS IoT. The MQTT test client is a useful tool to monitor and troubleshoot device connections.

> **Note**
>
> If you want to try more than one of these getting started tutorials or repeat the same tutorial, you should delete the thing object that you created from an earlier tutorial before you start another one. If you don't delete the thing object from an earlier tutorial, you will

need to use a different thing name for subsequent tutorials. This is because the thing name must be unique in your account and AWS Region.

For more information about AWS IoT Core, see What Is AWS IoT Core?

Set up AWS account

Before you use AWS IoT Core for the first time, complete the following tasks:

Topics

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

- Open the AWS IoT console

If you already have an AWS account and a user for yourself, you can use them and skip ahead to the section called "Open the AWS IoT console".

Open the AWS IoT console

Most of the console-oriented topics in this section start from the AWS IoT console. If you aren't already signed in to your AWS account, sign in, then open the AWS IoT console and continue to the next section to continue getting started with AWS IoT.

Interactive tutorial

The interactive tutorial shows the components of a simple IoT solution built on AWS IoT. The tutorial shows how IoT devices interact with AWS IoT Core services. This topic provides a preview of the AWS IoT Core interactive tutorial.

> **Note**
>
> The images in the console include animations that don't appear in the images in this tutorial.

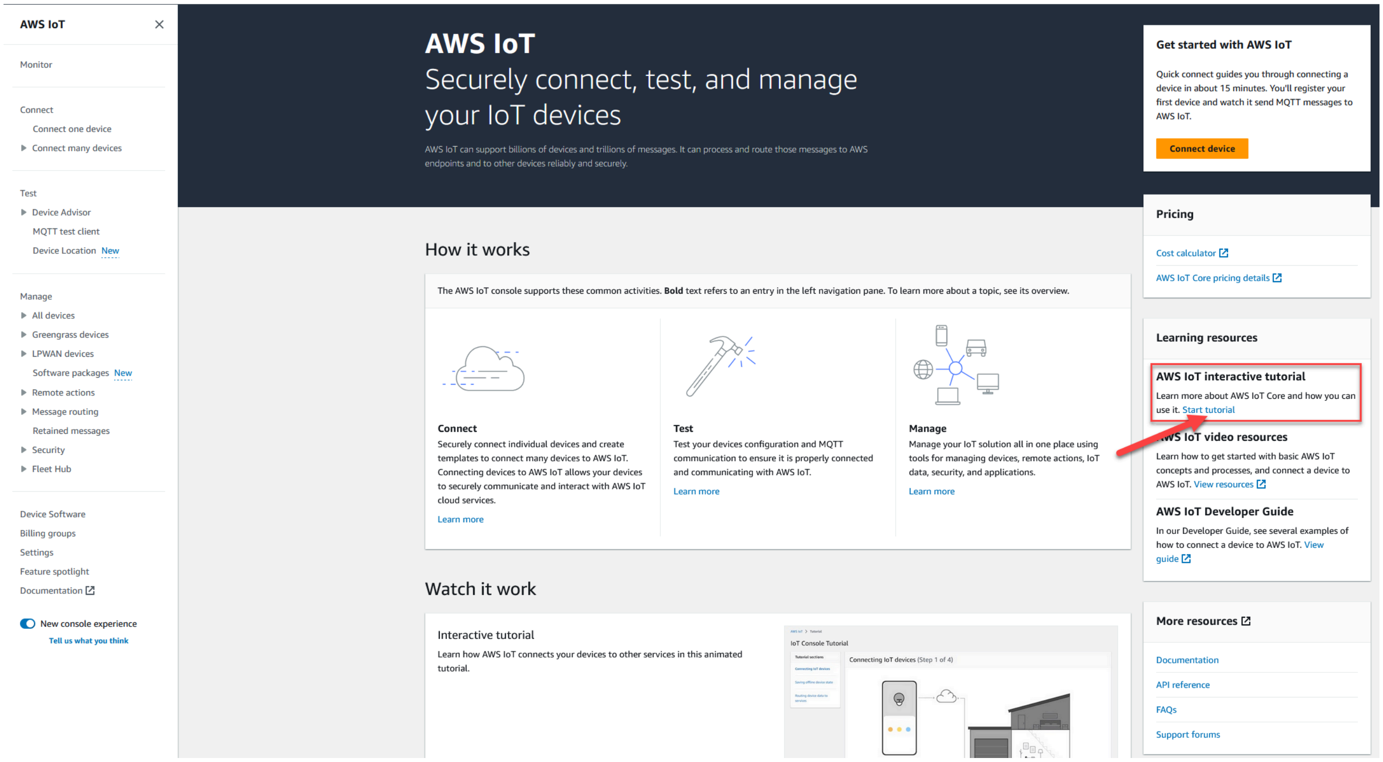
To run the demo, you must first the section called "Set up AWS account". The tutorial, however, doesn't require any AWS IoT resources, additional software, or any coding.

Expect to spend approximately 5-10 minutes on this demo. Giving yourself 10 minutes will allow more time to comprehend each of the steps.

To run the AWS IoT Core interactive tutorial

1. Open the [AWS IoT home page](https://console.aws.amazon.com/iot/home) in the AWS IoT console.

   On the **AWS IoT home page**, in the **Learning resources** window pane, choose **Start tutorial**.



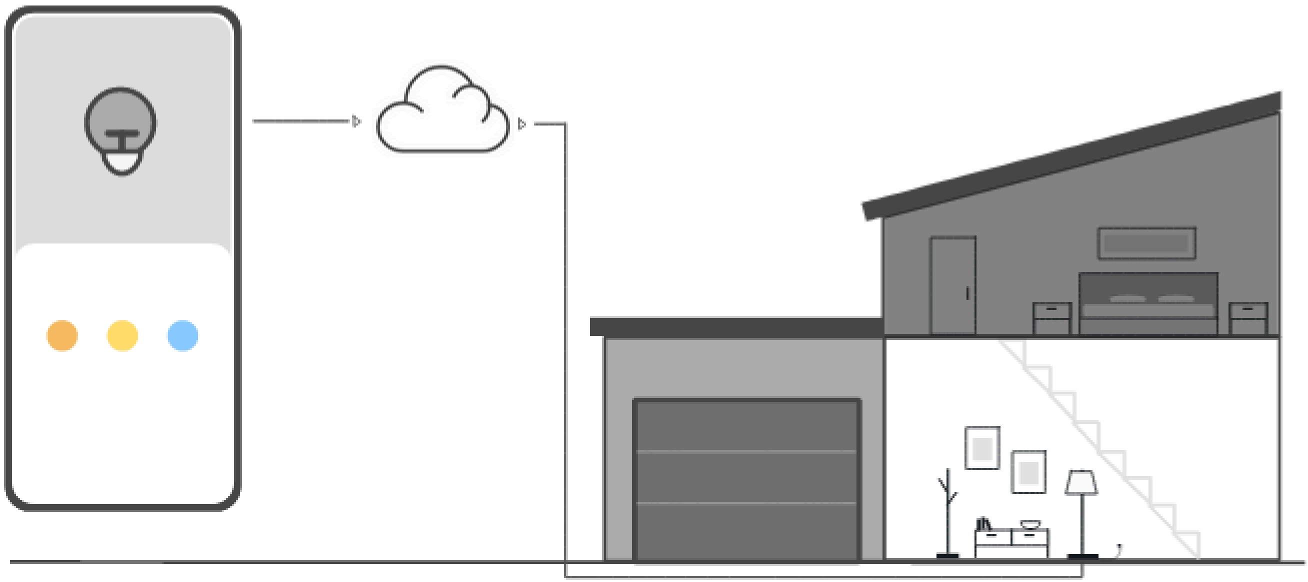
2. In the **AWS IoT Console Tutorial** page, review the tutorial sections and choose **Start section** when you're ready to continue.

The following sections describe how the AWS IoT Console Tutorial presents these AWS IoT Core features:

- Connecting IoT devices
- Saving offline device state
- Routing device data to services

Connecting IoT devices

Learn how IoT devices communicate with AWS IoT Core.

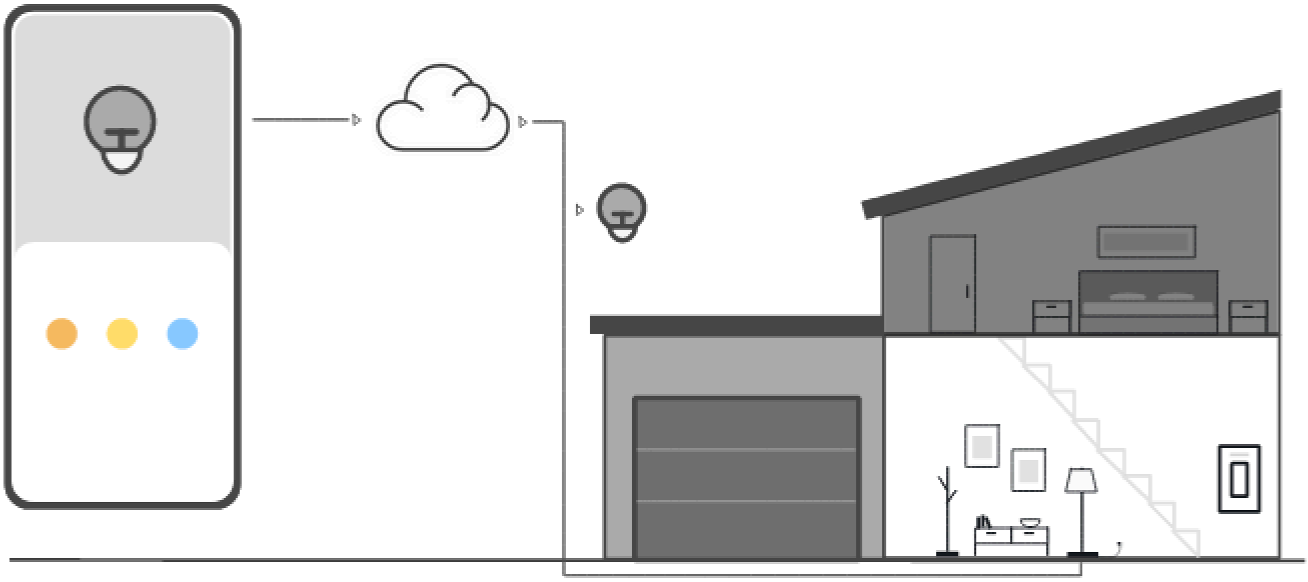


The animation in this step shows how two devices, the control device on the left and a smart lamp in the house on the right, connect and communicate with AWS IoT Core in the cloud. The animation shows the devices communicating with AWS IoT Core and reacting to the messages they receive.

For more information about connecting devices to AWS IoT Core, see Connect to AWS IoT Core.

Saving offline device state

Learn how AWS IoT Core saves device state for while a device or app is offline.



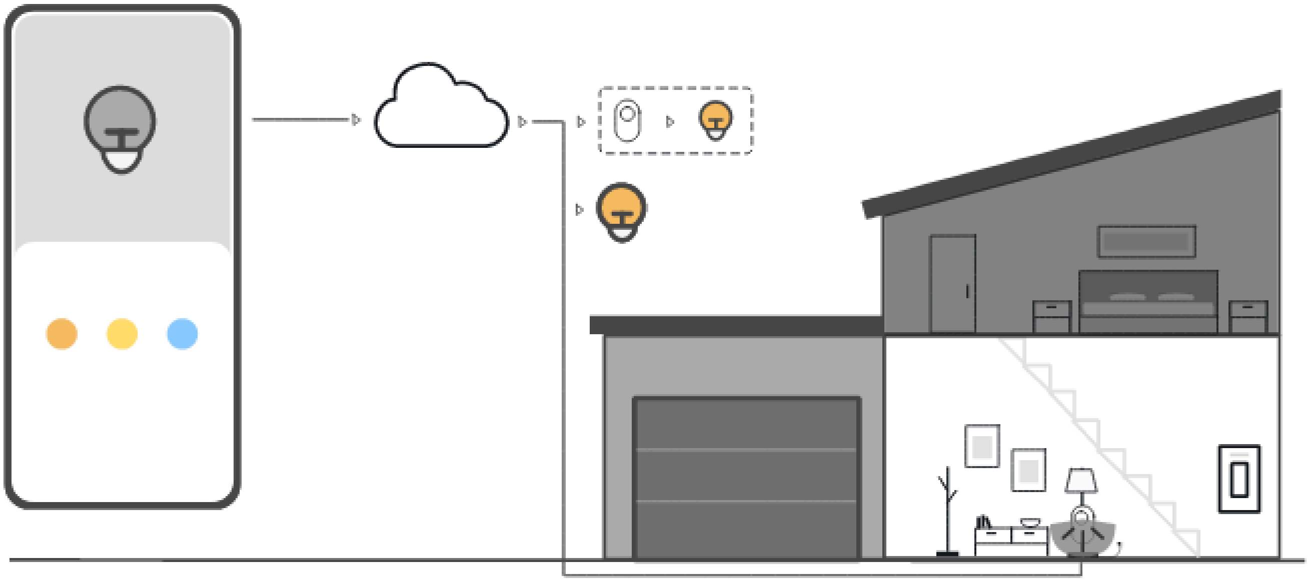
The animation in this step shows how the Device Shadow service in AWS IoT Core saves device state information for the control device and the smart lamp. While the smart lamp is offline, the Device Shadow saves commands from the control device.

When the smart lamp reconnects to AWS IoT Core, it retrieves those commands. When the control device is offline, the Device Shadow saves state information from the smart lamp. When the control device reconnects, it retrieves the current state of the smart lamp to update its display.

For more information about Device Shadows, see AWS IoT Device Shadow service.

Routing device data to services

Learn how AWS IoT Core sends device state to other AWS services.



The animation in this step shows how AWS IoT Core sends data from the devices to other AWS services by using AWS IoT rules. AWS IoT rules subscribe to specific messages from the devices, interpret the data in those messages, and route the interpreted data to other services. In this example, an AWS IoT rule interprets data from a motion sensor and sends commands to a Device Shadow, which then sends them to the smart bulb. As in the previous example, the Device Shadow stores the device-state info for the control device.

For more information about AWS IoT rules, see Rules for AWS IoT.

Try the AWS IoT Core quick connect tutorial

In this tutorial, you'll create your first thing object, connect a device to it, and watch it send MQTT messages.

You can expect to spend 15-20 minutes on this tutorial.

This tutorial is best for people who want to quickly get started with AWS IoT to see how it works in a limited scenario. If you're looking for an example that will get you started so that you can explore more features and services, try Explore AWS IoT Core in hands-on tutorials.

In this tutorial, you'll download and run software on a device that connects to a *thing resource* in AWS IoT Core as part of a very small IoT solution. The device can be an IoT device, such as a Raspberry Pi, or it can also be a computer that is running Linux, OS and OSX, or Windows. If you're looking to connect a Long Range WAN (LoRaWAN) device to AWS IoT, refer to the tutorial >Connecting devices and gateways to AWS IoT Core for LoRaWAN.

If your device supports a browser that can run the AWS IoT console, we recommend you complete this tutorial on that device.

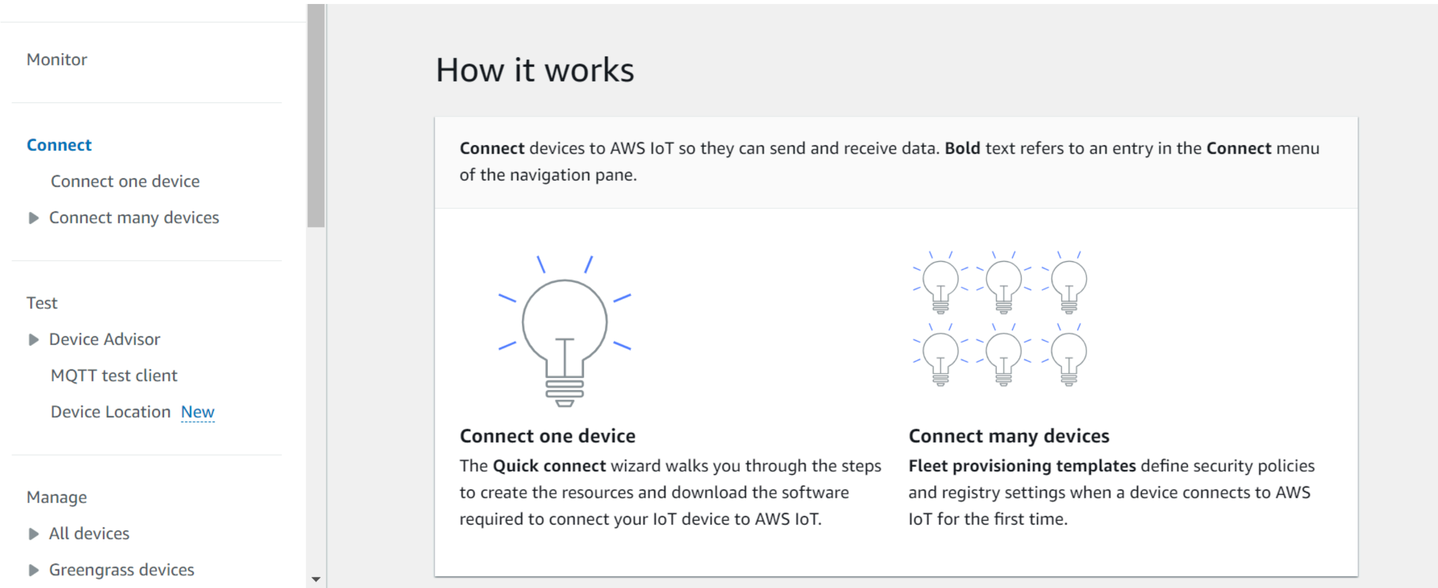
> **Note**
>
> If your device doesn't have a compatible browser, follow this tutorial on a computer. When the procedure asks you to download the file, download it to your computer, and then transfer the downloaded file to your device by using Secure Copy (SCP) or a similar process.

The tutorial requires your IoT device to communicate with port 8443 on your AWS account's device data endpoint. To test whether it can access that port, try the procedures in Test connectivity with your device data endpoint.

Step 1. Start the tutorial

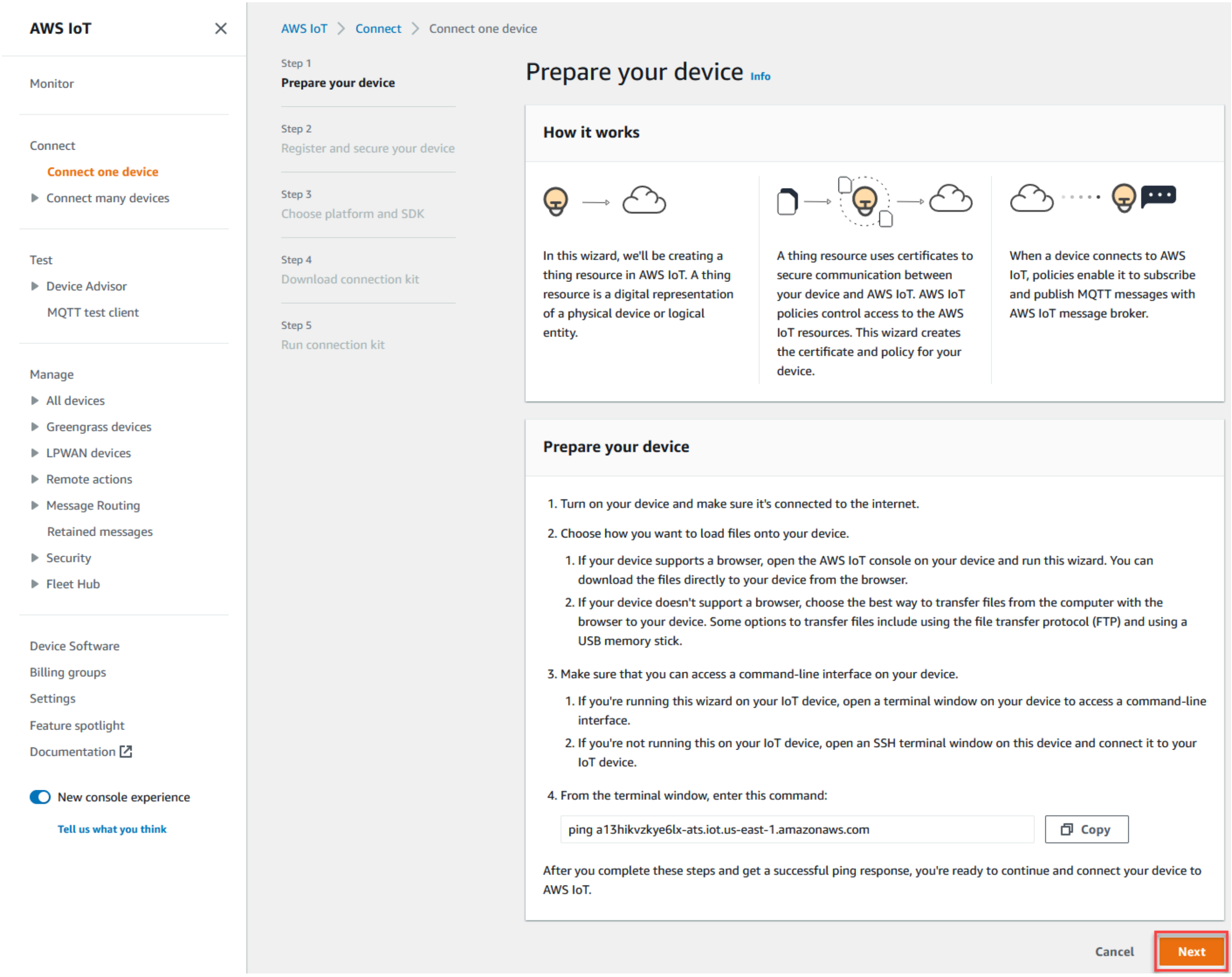
If possible, complete this procedure on your device; otherwise, be ready to transfer a file to your device later in this procedure.

To start the tutorial, sign in to the AWS IoT console. In the AWS IoT console home page, on the left, choose **Connect** and then choose **Connect one device**.



Step 2. Create a thing object

1. In the **Prepare your device** section, follow the on-screen instructions to prepare your device for connecting to AWS IoT.



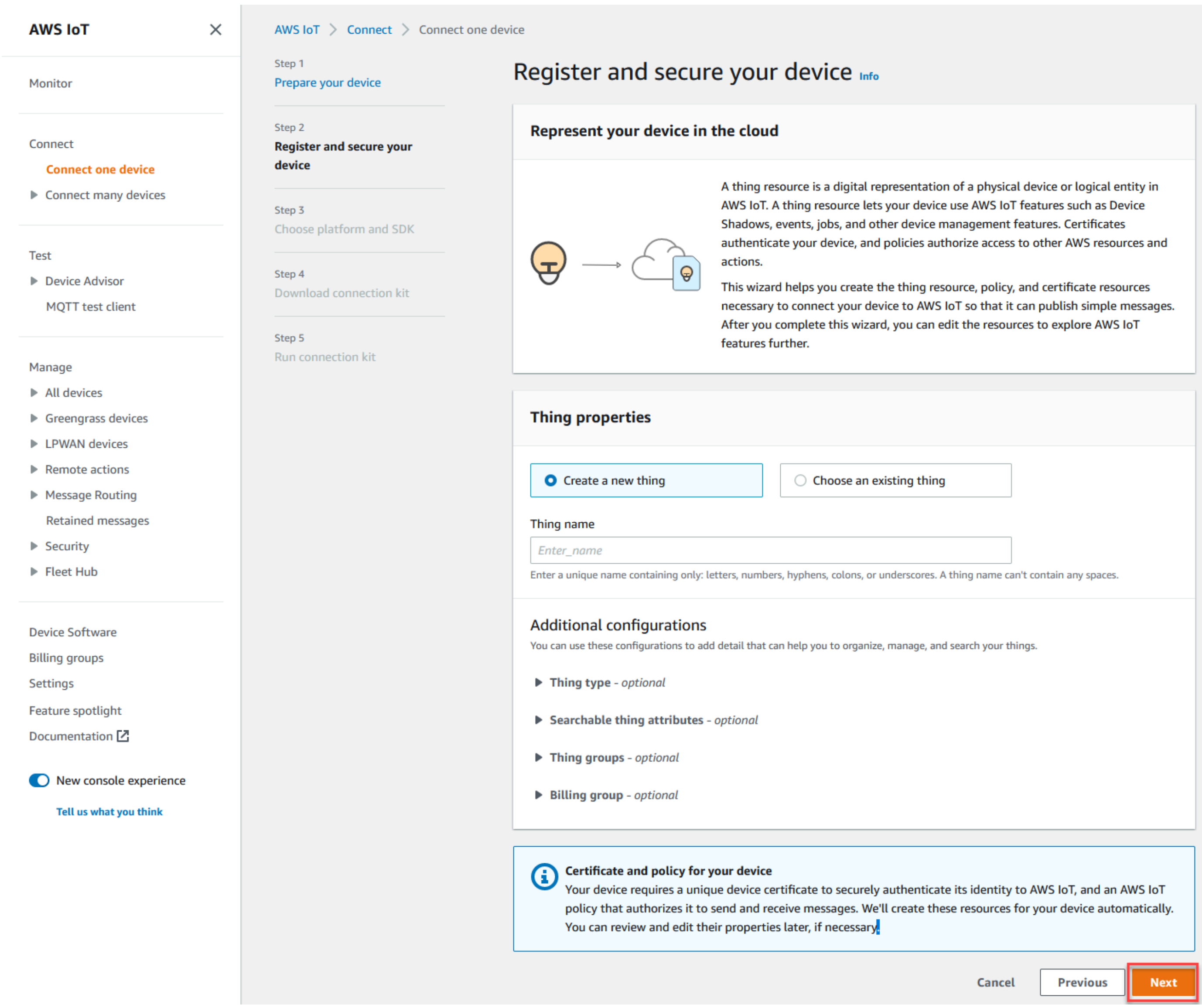
2. In the **Register and secure your device** section, choose **Create a new thing** or **Choose an existing thing**. In the **Thing name** field, enter the name for your thing object. The thing name used in this example is **`TutorialTestThing`**

> ⚠ **Important**
>
> Double-check your thing name before you continue.

A thing name can't be changed after the thing object is created. If you want to change a thing name, you must create a new thing object with the correct thing name and then delete the one with the incorrect name.

In the **Additional configurations** section, customize your thing resource further using the optional configurations listed.

After you provide your thing object a name and select any additional configurations, choose **Next**.

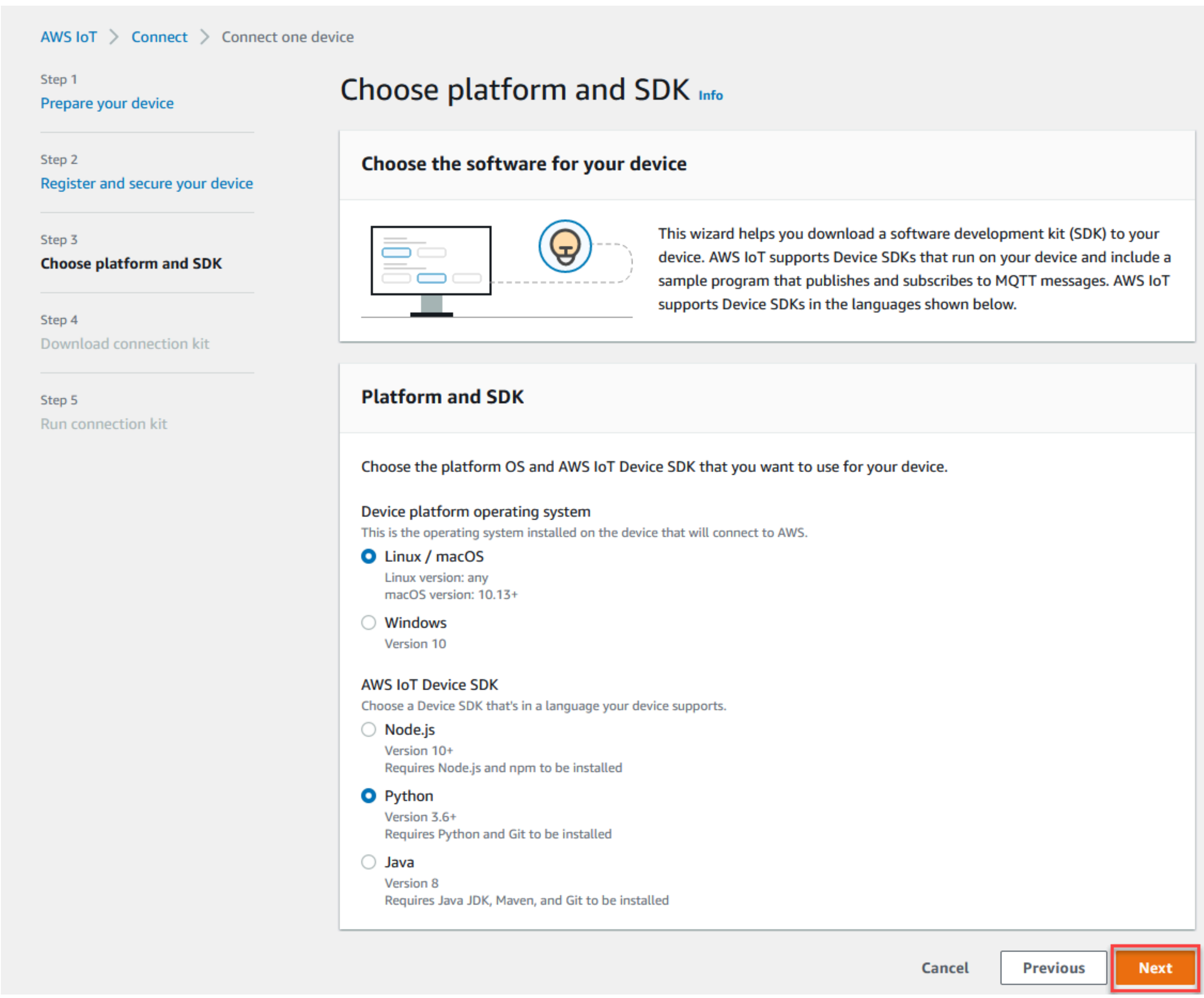


3. In the **Choose platform and SDK** section, choose the platform and the language of the AWS IoT Device SDK that you want to use. This example uses the Linux/OSX platform and the

Python SDK. Make sure that you have python3 and pip3 installed on your target device before you continue to the next step.

> **Note**
>
> Be sure to check the list of prerequisite software required by your chosen SDK at the bottom of the console page.
> You must have the required software installed on your target computer before you continue to the next step.

After you choose the platform and device SDK language, choose **Next**.

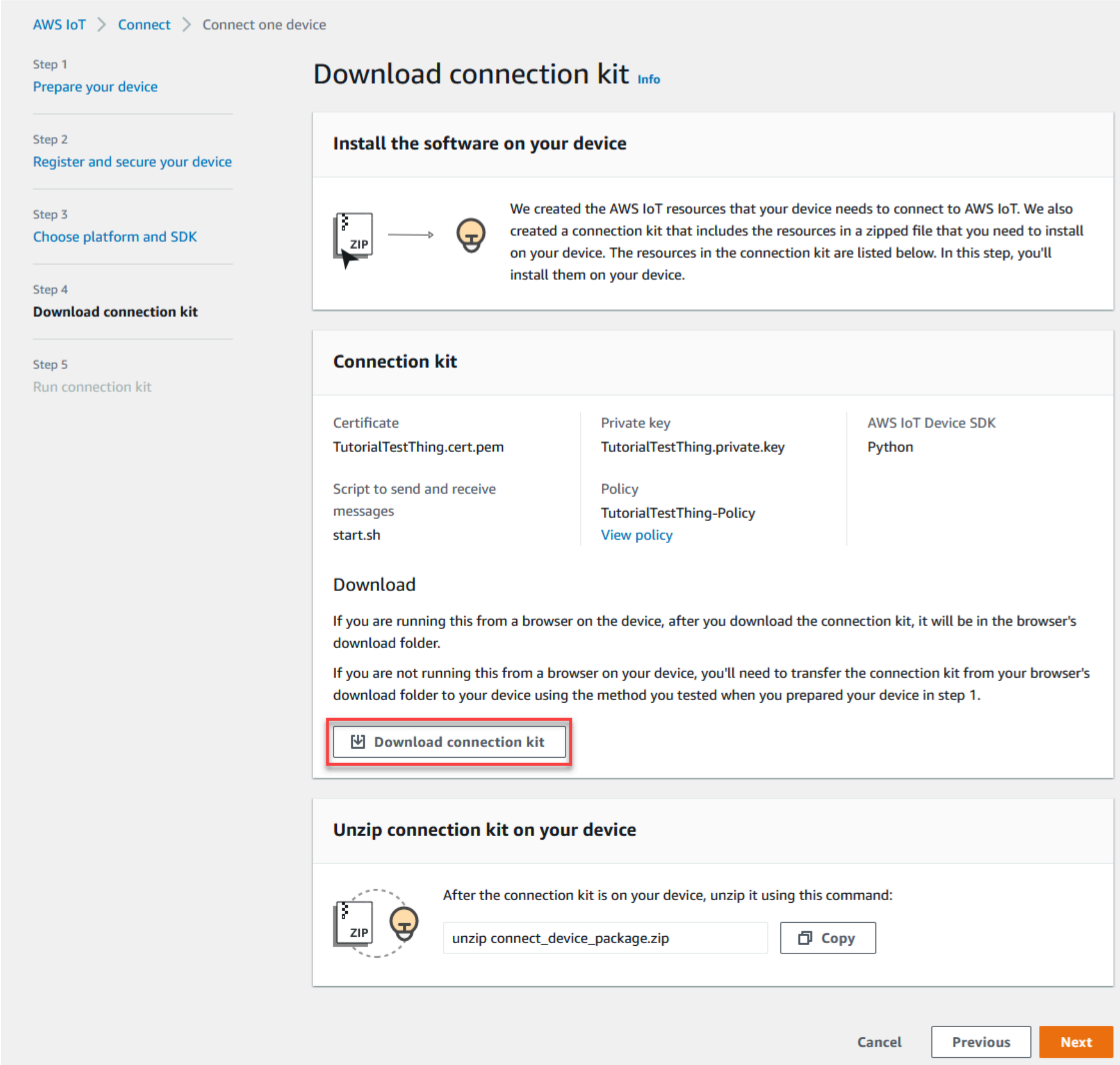


Step 3. Download files to your device

This page appears after AWS IoT has created the connection kit, which includes the following files and resources that your device requires:

- The thing's certificate files used to authenticate the device
- A policy resource to authorize your thing object to interact with AWS IoT
- The script to download the AWS Device SDK and run the sample program on your device

1. When you're ready to continue, choose the **Download connection kit for** button to download the connection kit for the platform that you chose earlier.



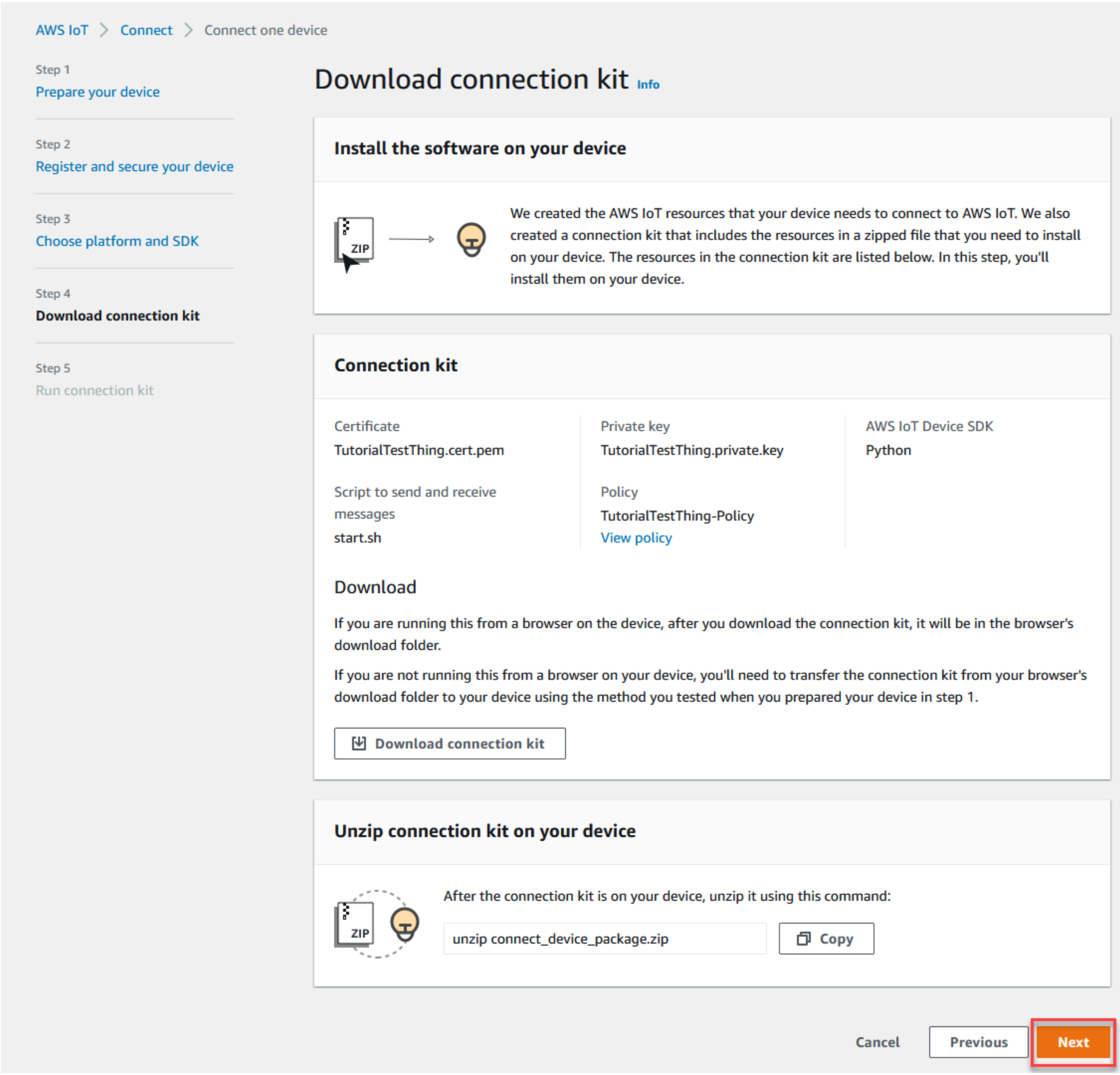
2. If you're running this procedure on your device, save the connection kit file to a directory from which you can run command line commands.

   If you're not running this procedure on your device, save the connection kit file to a local directory and then transfer the file to your device.

3. In the **Unzip connection kit on your device** section, enter **unzip connect_device_package.zip** in the directory where the connection kit files are located.

If you're using a Windows PowerShell command window and the **unzip** command doesn't work, replace **unzip** with **expand-archive**, and try the command line again.

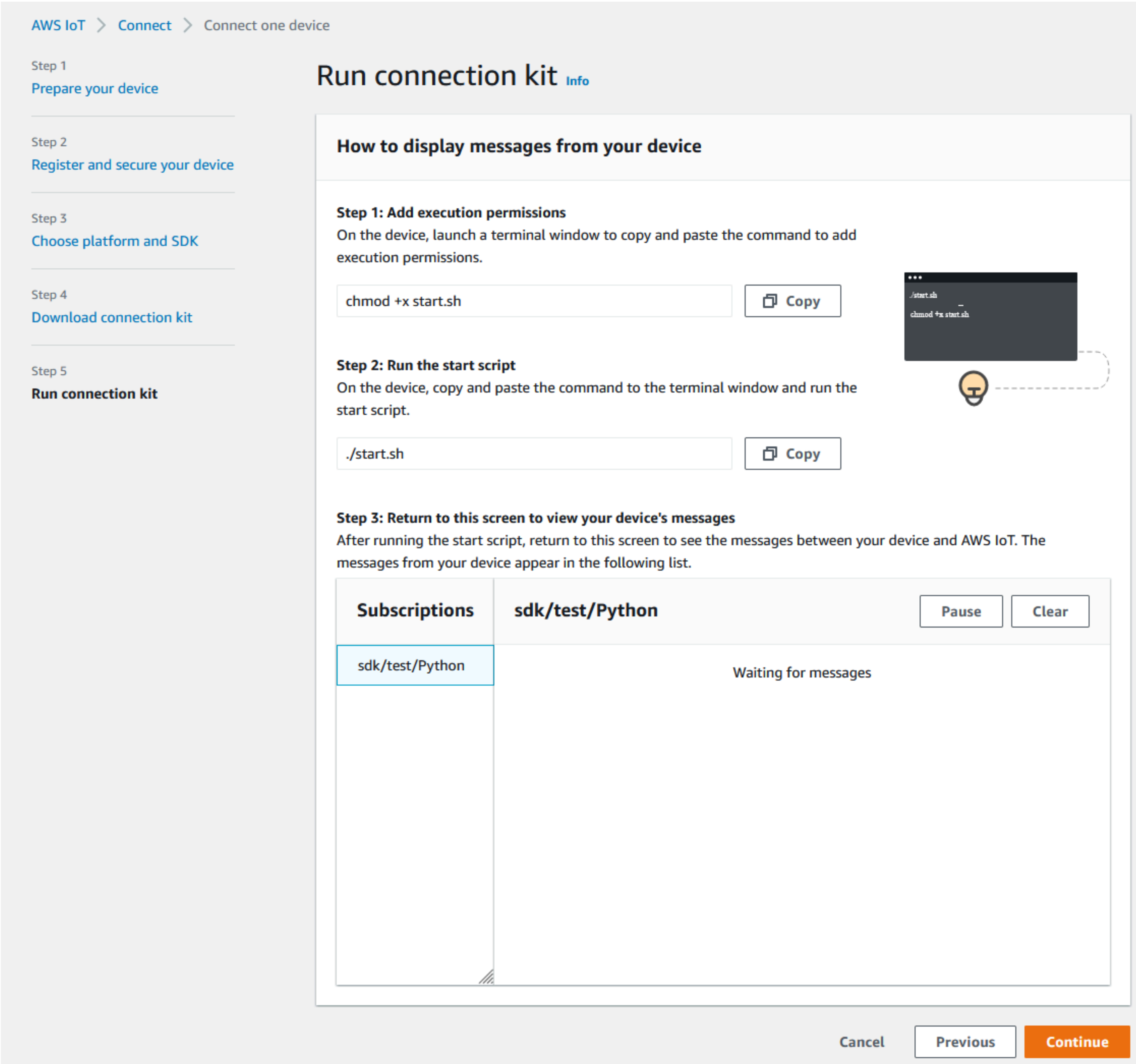
4. After you have the connection kit file on the device, continue the tutorial by choosing **Next**.



Step 4. Run the sample

You do this procedure in a terminal or command window on your device while you follow the directions displayed in the console. The commands you see in the console are for the operating system you chose in the section called “Step 2. Create a thing object”. Those shown here are for the Linux/OSX operating systems.

1. In a terminal or command window on your device, in the directory with the connection kit file, perform the steps shown in the AWS IoT console.



2. After you enter the command from **Step 2** in the console, you should see an output in the device's terminal or command window that is similar to the following. This output is from the messages the program is sending to and then receiving back from AWS IoT Core.

```
Running pub/sub sample application...
Connecting to a13hikvzkye6lx-ats.iot.us-east-1.amazonaws.com with client ID 'basicPubSub'...
Connected!
Subscribing to topic 'sdk/test/Python'...
Subscribed with QoS.AT_LEAST_ONCE
Sending messages until program killed
Publishing message to topic 'sdk/test/Python': Hello World! [1]
Received message from topic 'sdk/test/Python': b'"Hello World! [1]"'
Publishing message to topic 'sdk/test/Python': Hello World! [2]
Received message from topic 'sdk/test/Python': b'"Hello World! [2]"'
Publishing message to topic 'sdk/test/Python': Hello World! [3]
Received message from topic 'sdk/test/Python': b'"Hello World! [3]"'
```

While the sample program is running, the test message `Hello World!` will appear as well. The test message appears in the terminal or command window on your device.

> **Note**
>
> For more information about topic subscription and publish, see the example code of your chosen SDK.

3. To run the sample program again, you can repeat the commands from **Step 2** in the console of this procedure.

4. (Optional) If you want to see the messages from your IoT client in the AWS IoT console, open the MQTT test client on the **Test** page of the AWS IoT console. If you chose Python SDK, then in the **MQTT test client**, in **Topic filter**, enter the topic, such as **`sdk/test/python`** to subscribe to the messages from your device. The topic filters are case sensitive and depend on the programming language of the SDK you chose in **Step 1**. For more information about topic subscription and publish, see the code example of your chosen SDK.

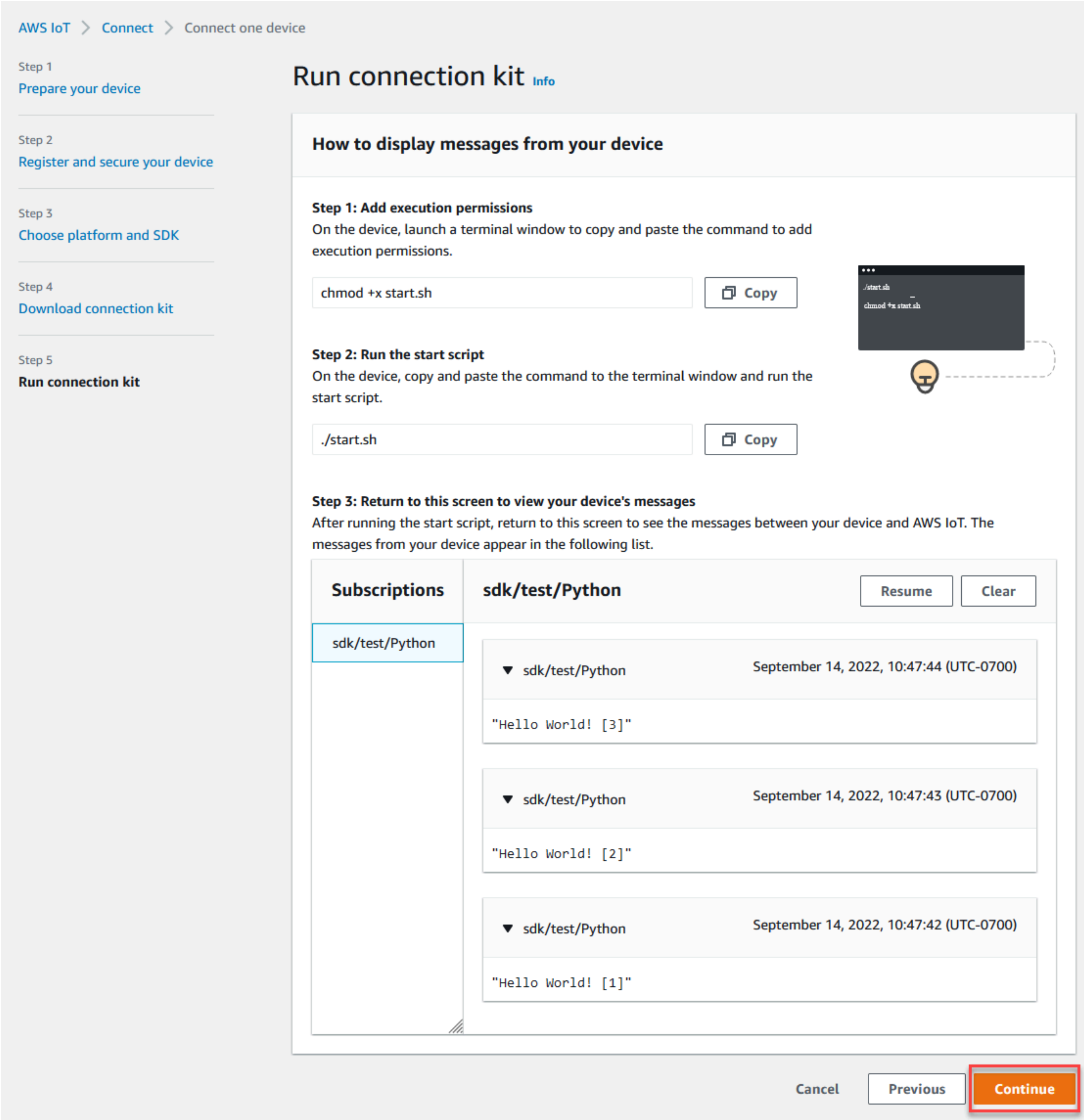
5. After you subscribe to the test topic, run **./start.sh** on your device. For more information, see the section called "View MQTT messages with the AWS IoT MQTT client".

   After you run **./start.sh**, messages appear in the MQTT client, similar to the following:

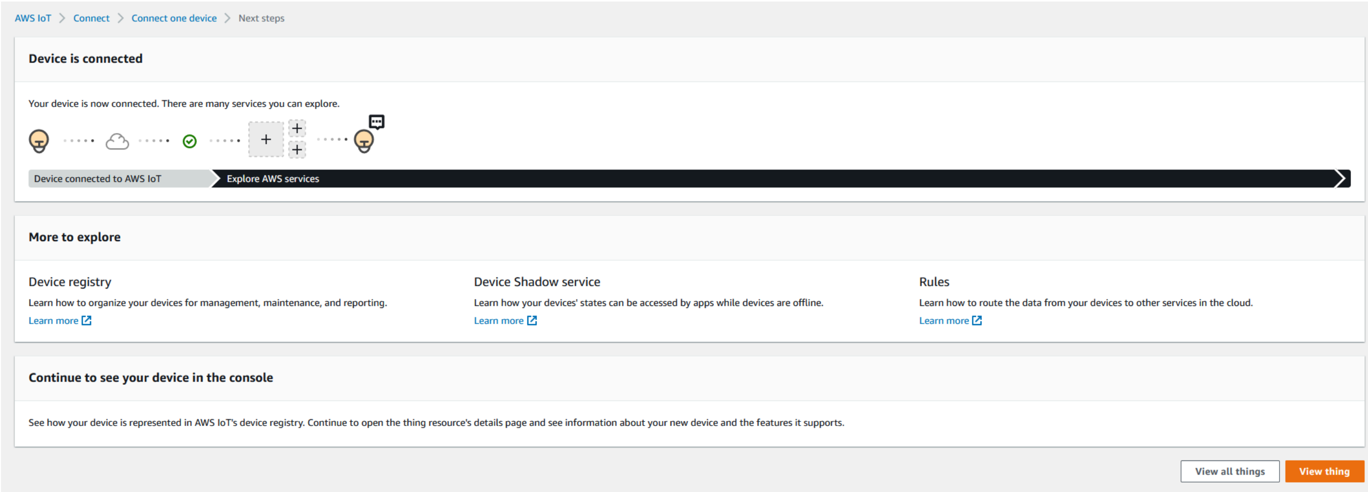
```
{
  "message": "Hello World!" [1]
}
```

   The `sequence` number encased in `[]` increments by one each time a new `Hello World!` message is received and stops when you end the program.

6. To finish the tutorial and see a summary, in the AWS IoT console, choose **Continue**.



7. A summary of your AWS IoT quick connect tutorial will now appear.



Step 5. Explore further

Here are some ideas to explore AWS IoT further after you complete the quick start.

- **View MQTT messages in the MQTT test client**

  From the AWS IoT console, you can open the MQTT client on the **Test** page of the AWS IoT console. In the **MQTT test client**, subscribe to **#**, and then, on your device, run the program **./start.sh** as described in the previous step. For more information, see the section called "View MQTT messages with the AWS IoT MQTT client".

- **Run tests on your devices with Device Advisor**

  Use Device Advisor to test if your devices can securely and reliably connect to, and interact with, AWS IoT.

- **the section called "Interactive tutorial"**

  To start the interactive tutorial, from the **Learn** page of the AWS IoT console, in the **See how AWS IoT works** tile, choose **Start the tutorial**.

- **Get ready to explore more tutorials**

  This quick start gives you just a sample of AWS IoT. If you want to explore AWS IoT further and learn about the features that make it a powerful IoT solution platform, start preparing your development platform by Explore AWS IoT Core in hands-on tutorials.

Test connectivity with your device data endpoint

This topic describes how to test a device's connection with your account's *device data endpoint*, the endpoint that your IoT devices use to connect to AWS IoT.

Perform these procedures on the device that you want to test or by using an SSH terminal session connected to the device you want to test.

To test a device's connectivity with your device data endpoint.

- Find your device data endpoint
- Test the connection quickly
- Get the app to test the connection to your device data endpoint and port
- Test the connection to your device data endpoint and port

Find your device data endpoint

This procedure explains how to find your device data endpoint in the AWS IoT console for testing the connection to your IoT device.

To find your device data endpoint

1. In the AWS IoT console, in the **Connect** section, go to **Domain Configurations**.
2. In the **Domain Configurations** page, go to the **Domain configurations** container, and copy the **Domain name**. Your endpoint value is unique to your AWS account and is similar to this example: `a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com`.
3. Save your device data endpoint to use in the following procedures.

Test the connection quickly

This procedure tests general connectivity with your device data endpoint, but it doesn't test the specific port that your devices will use. This test uses a common program and is usually sufficient to find out if your devices can connect to AWS IoT.

If you want to test connectivity with the specific port that your devices will use, skip this procedure and continue to Get the app to test the connection to your device data endpoint and port.

To test the device data endpoint quickly

1. In a terminal or command line window on your device, replace the sample device data endpoint (*a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com*) with the device data endpoint for your account, and then enter this command.

   Linux

   ```
   ping -c 5 a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com
   ```

   Windows

   ```
   ping -n 5 a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com
   ```

2. If `ping` displays an output similar to the following, it connected to your device data endpoint successfully. While it didn't communicate with AWS IoT directly, it did find the server and it's likely that AWS IoT is available through this endpoint.

   ```
   PING a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com (xx.xx.xxx.xxx) 56(84) bytes of
    data.
   64 bytes from ec2-EXAMPLE-218.eu-west-1.compute.amazonaws.com (xx.xx.xxx.xxx):
    icmp_seq=1 ttl=231 time=127 ms
   64 bytes from ec2-EXAMPLE-218.eu-west-1.compute.amazonaws.com (xx.xx.xxx.xxx):
    icmp_seq=2 ttl=231 time=127 ms
   64 bytes from ec2-EXAMPLE-218.eu-west-1.compute.amazonaws.com (xx.xx.xxx.xxx):
    icmp_seq=3 ttl=231 time=127 ms
   64 bytes from ec2-EXAMPLE-218.eu-west-1.compute.amazonaws.com (xx.xx.xxx.xxx):
    icmp_seq=4 ttl=231 time=127 ms
   64 bytes from ec2-EXAMPLE-218.eu-west-1.compute.amazonaws.com (xx.xx.xxx.xxx):
    icmp_seq=5 ttl=231 time=127 ms
   ```

   If you are satisfied with this result, you can stop testing here.

   If you want to test the connectivity with the specific port used by AWS IoT, continue to Get the app to test the connection to your device data endpoint and port.

3. If `ping` didn't return a successful output, check the endpoint value to make sure you have the correct endpoint and check the device's connection with the internet.

Get the app to test the connection to your device data endpoint and port

A more thorough connectivity test can be performed by using `nmap`. This procedure tests to see if `nmap` is installed on your device.

To check for nmap on the device

1. In a terminal or command line window on the device you want to test, enter this command to see if `nmap` is installed.

   ```
   nmap --version
   ```

2. If you see an output similar to the following, `nmap` is installed and you can continue to [the section called “Test the connection to your device data endpoint and port”](#).

   ```
   Nmap version 6.40 ( http://nmap.org )
   Platform: x86_64-koji-linux-gnu
   Compiled with: nmap-liblua-5.2.2 openssl-1.0.2k libpcre-8.32 libpcap-1.5.3 nmap-
   libdnet-1.12 ipv6
   Compiled without:
   Available nsock engines: epoll poll select
   ```

3. If you don't see a response similar to the one shown in the preceding step, you must install `nmap` on the device. Choose the procedure for your device's operating system.

Linux

This procedure requires that you have permission to install software on the computer.

To install nmap on your Linux computer

1. In a terminal or command line window on your device, enter the command that corresponds to the version of Linux it's running.

   a. Debian or Ubuntu:

      ```
      sudo apt install nmap
      ```

   b. CentOS or RHEL:

      ```
      sudo yum install nmap
      ```

2. Test the installation with this command:

```
nmap --version
```

3. If you see an output similar to the following, nmap is installed and you can continue to [the section called “Test the connection to your device data endpoint and port”](#).

```
Nmap version 6.40 ( http://nmap.org )
Platform: x86_64-koji-linux-gnu
Compiled with: nmap-liblua-5.2.2 openssl-1.0.2k libpcre-8.32 libpcap-1.5.3 nmap-libdnet-1.12 ipv6
Compiled without:
Available nsock engines: epoll poll select
```

macOS

This procedure requires that you have permission to install software on the computer.

To install nmap on your macOS computer

1. In a browser, open https://nmap.org/download#macosx and download the **latest stable** installer.

   When prompted, select **Open with DiskImageInstaller**.
2. In the installation window, move the package to the **Applications** folder.
3. In the **Finder**, locate the `nmap-xxxx-mpkg` package in the **Applications** folder. **Ctrl-click** the on package and select **Open** to open the package.
4. Review the security dialog box. If you are ready to install **nmap**, choose **Open** to install **nmap**.
5. In **Terminal**, test the installation with this command.

```
nmap --version
```

6. If you see an output similar to the following, nmap is installed and you can continue to [the section called “Test the connection to your device data endpoint and port”](#).

```
Nmap version 7.92 ( https://nmap.org )
Platform: x86_64-apple-darwin17.7.0
```

```
Compiled with: nmap-liblua-5.3.5 openssl-1.1.1k nmap-libssh2-1.9.0 libz-1.2.11
 nmap-libpcre-7.6 nmap-libpcap-1.9.1 nmap-libdnet-1.12 ipv6 Compiled without:
Available nsock engines: kqueue poll select
```

Windows

This procedure requires that you have permission to install software on the computer.

To install nmap on your Windows computer

1. In a browser, open https://nmap.org/download#windows and download the **latest stable** release of the setup program.

   If prompted, choose **Save file**. After the file is downloaded, open it from the downloads folder.
2. After the setup file finishes downloading, open downloaded **nmap-xxxx-setup.exe** to install the app.
3. Accept the default settings as the program installs.

   You don't need the Npcap app for this test. You can deselect that option if you don't want to install it.
4. In **Command**, test the installation with this command.

   ```
   nmap --version
   ```
5. If you see an output similar to the following, `nmap` is installed and you can continue to the section called “Test the connection to your device data endpoint and port”.

   ```
   Nmap version 7.92 ( https://nmap.org )
   Platform: i686-pc-windows-windows
   Compiled with: nmap-liblua-5.3.5 openssl-1.1.1k nmap-libssh2-1.9.0 nmap-
   libz-1.2.11 nmap-libpcre-7.6 Npcap-1.50 nmap-libdnet-1.12 ipv6
   Compiled without:
   Available nsock engines: iocp poll select
   ```

Test the connection to your device data endpoint and port

This procedure tests your IoT device's connection to your device data endpoint using your selected port.

To test your device data endpoint and port

1. In a terminal or command line window on your device, replace the sample device data endpoint (*a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com*) with the device data endpoint for your account, and then enter this command.

```
nmap -p 8443 a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com
```

2. If nmap displays an output similar to the following, nmap was able to connect successfully to your device data endpoint at the selected port.

```
Starting Nmap 7.92 ( https://nmap.org ) at 2022-02-18 16:23 Pacific Standard Time
Nmap scan report for a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com
 (xx.xxx.147.160)
Host is up (0.036s latency).
Other addresses for a3qEXAMPLEsffp-ats.iot.eu-west-1.amazonaws.com (not scanned):
 xx.xxx.134.144 xx.xxx.55.139 xx.xxx.110.235 xx.xxx.174.233 xx.xxx.74.65
 xx.xxx.122.179 xx.xxx.127.126
rDNS record for xx.xxx.147.160: ec2-EXAMPLE-160.eu-west-1.compute.amazonaws.com

PORT     STATE SERVICE
8443/tcp open  https-alt
MAC Address: 00:11:22:33:44:55 (Cimsys)

Nmap done: 1 IP address (1 host up) scanned in 0.91 seconds
```

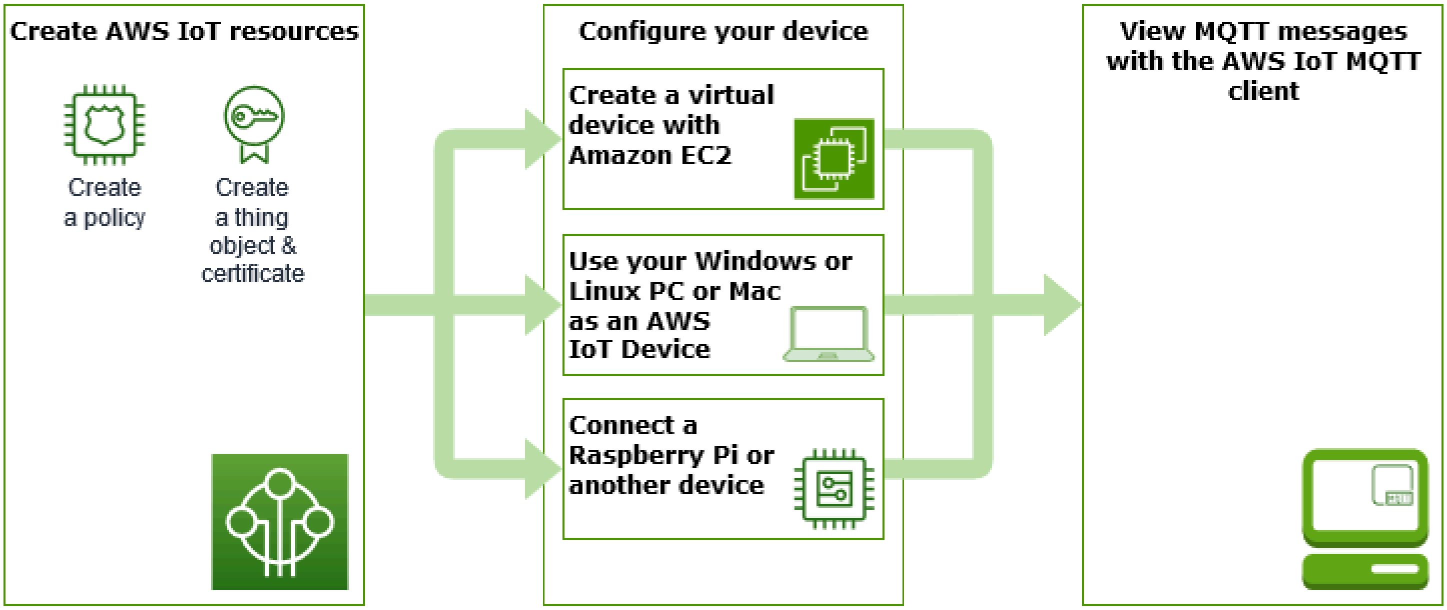
3. If nmap didn't return a successful output, check the endpoint value to make sure you have the correct endpoint and check your device's connection with the internet.

You can test other ports on your device data endpoint, such as port 443, the primary HTTPS port, by replacing the port used in step 1, **8443**, with the port that you want to test.

Explore AWS IoT Core in hands-on tutorials

In this tutorial, you'll install the software and create the AWS IoT resources necessary to connect a device to AWS IoT Core so that it can send and receive MQTT messages with AWS IoT Core. You'll see the messages in the MQTT client in the AWS IoT console.

You can expect to spend 20-30 minutes on this tutorial. If you are using an IoT device or a Raspberry Pi, this tutorial might take longer if, for example, you need to install the operating system and configure the device.



This tutorial is best for developers who want to get started with AWS IoT Core so they can continue to explore more advanced features, such as the rules engine and shadows. This tutorial prepares you to continue learning about AWS IoT Core and how it interacts with other AWS services by explaining the steps in greater detail than the quick start tutorial. If you are looking for just a quick, *Hello World*, experience, try the Try the AWS IoT Core quick connect tutorial.

After setting up your AWS account and AWS IoT console, you'll follow these steps to see how to connect a device and have it send messages to AWS IoT Core.

Next steps

- Choose which device option is the best for you
- the section called "Create AWS IoT resources" if you are not going to create a virtual device with Amazon EC2
- the section called "Configure your device"

- the section called "View MQTT messages with the AWS IoT MQTT client"

For more information about AWS IoT Core, see What Is AWS IoT Core?

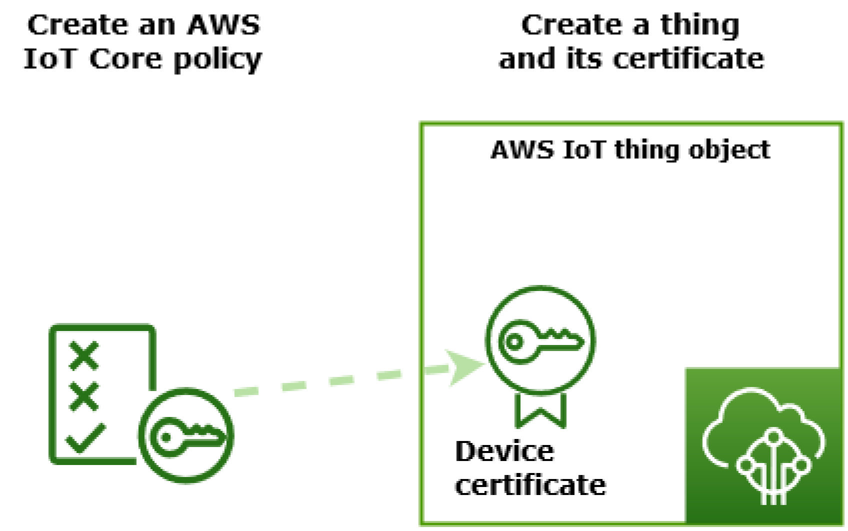
Which device option is best for you?

If you're not sure which option to pick, use the following list of each option's advantages and disadvantages to help you decide which one is best for you.

Option	This might be a good option if:	This might not be a good option if:
the section called "Create a virtual device with Amazon EC2"	• You don't have your own device to test. • You don't want to install any software on your own system. • You want to test on a Linux OS.	• You're not comfortab le using command-line commands. • You don't want to incur any additional AWS charges. • You don't want to test on a Linux OS.
the section called "Use your Windows or Linux PC or Mac as an AWS IoT device"	• You don't want to incur any additional AWS charges. • You don't want to configure any additional devices.	• You don't want to install any software on your personal computer. • You want a more represent ative test platform.
the section called "Connect a Raspberry Pi or other device"	• You want to test AWS IoT with an actual device. • You already have a device to test with. • You have experience integrating hardware into systems.	• You don't want to buy or configure a device just to try it out. • You want to test AWS IoT as simply as possible, for now.

Create AWS IoT resources

In this tutorial, you'll create the AWS IoT resources that a device requires to connect to AWS IoT Core and exchange messages.



1. Create an AWS IoT policy document, which will authorize your device to interact with AWS IoT services.
2. Create a thing object in AWS IoT and its X.509 device certificate, and then attach the policy document. The thing object is the virtual representation of your device in the AWS IoT registry. The certificate authenticates your device to AWS IoT Core, and the policy document authorizes your device to interact with AWS IoT.

> **Note**
>
> If you are planning to the section called "Create a virtual device with Amazon EC2", you can skip this page and continue to the section called "Configure your device". You will create these resources when you create your virtual thing.

This tutorial uses the AWS IoT console to create the AWS IoT resources. If your device supports a web browser, it might be easier to run this procedure on the device's web browser because you will be able to download the certificate files directly to your device. If you run this procedure on another computer, you will need to copy the certificate files to your device before they can be used by the sample app.

Create an AWS IoT policy

Devices use an X.509 certificate to authenticate with AWS IoT Core. The certificate has AWS IoT policies attached to it. These policies determine which AWS IoT operations, such as subscribing or publishing to MQTT topics, the device is permitted to perform. Your device presents its certificate when it connects and sends messages to AWS IoT Core.

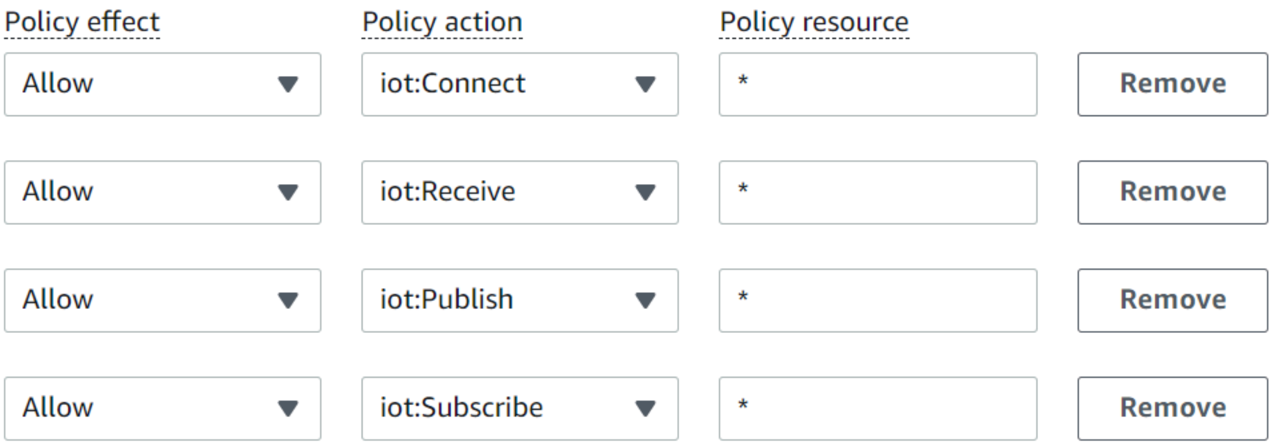
Follow the steps to create a policy that allows your device to perform the AWS IoT operations necessary to run the example program. You must create the AWS IoT policy before you can attach it to the device certificate, which you'll create later.

To create an AWS IoT policy

1. In the AWS IoT console, in the left menu, choose **Security** and then choose **Policies**.
2. On the **You don't have a policy yet** page, choose **Create policy**.

   If your account has existing policies, choose **Create policy**.
3. On the **Create policy** page:
   1. In the **Policy properties** section, in the **Policy name** field, enter a name for the policy (for example, **`My_Iot_Policy`**). Don't use personally identifiable information in your policy names.
   2. In the **Policy document** section, create the policy statements that grant or deny resources access to AWS IoT Core operations. To create a policy statement that grants all clients to perform **`iot:Connect`**, follow these steps:
      - In the **Policy effect** field, choose **Allow**. This allows all clients that have this policy attached to their certificate to perform the action listed in the **Policy action** field.
      - In the **Policy action** field, choose a policy action such as **`iot:Connect`**. Policy actions are the actions that your device needs permission to perform when it runs the example program from the Device SDK.
      - In the **Policy resource** field, enter a resource Amazon Resource Name (ARN) or **`*`**. A **`*`** to select any client (device).

      To create the policy statements for **`iot:Receive`**, **`iot:Publish`**, and **`iot:Subscribe`**, choose **Add new statement** and repeat the steps.



> **Note**
>
> In this quick start, the wildcard (*) character is used for simplicity. For higher security, you should restrict which clients (devices) can connect and publish messages by specifying a client ARN instead of the wildcard character as the resource. Client ARNs follow this format: `arn:aws:iot:your-region:your-aws-account:client/my-client-id`.
> However, you must first create the resource (such as a client device or thing shadow) before you can assign its ARN to a policy. For more information, see [AWS IoT Core action resources](#).

4. After you've entered the information for your policy, choose **Create**.

For more information, see How AWS IoT works with IAM.

Create a thing object

Devices connected to AWS IoT Core are represented by *thing objects* in the AWS IoT registry. A *thing object* represents a specific device or logical entity. It can be a physical device or sensor (for example, a light bulb or a light switch on the wall). It can also be a logical entity, like an instance of an application or physical entity that doesn't connect to AWS IoT, but is related to other devices that do (for example, a car that has engine sensors or a control panel).

To create a thing in the AWS IoT console

1. In the AWS IoT console, in the left menu, choose **All devices** and then choose **Things**.
2. On the **Things** page, choose **Create things**.

3. On the **Create things** page, choose **Create a single thing**, then choose **Next**.
4. On the **Specify thing properties** page, for **Thing name**, enter a name for your thing, such as **`MyIotThing`**.

   Choose thing names carefully, because you can't change a thing name later.

   To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.

   > **Note**
   >
   > Do not use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.

5. Keep the rest of the fields on this page empty. Choose **Next**.
6. On the **Configure device certificate - *optional*** page, choose **Auto-generate a new certificate (recommended)**. Choose **Next**.
7. On the **Attach policies to certificate - *optional*** page, select the policy you created in the previous section. In that section, the policy was named, **`My_Iot_Policy`**. Choose **Create thing**.
8. On the **Download certificates and keys** page:

   1. Download each of the certificate and key files and save them for later. You'll need to install these files on your device.

      When you save your certificate files, give them the names in the following table. These are the file names used in later examples.

      Certificate file names

File	File path
Private key	`private.pem.key`
Public key	*(not used in these examples)*
Device certificate	`device.pem.crt`
Root CA certificate	`Amazon-root-CA-1.pem`

2. To download the root CA file for these files, choose the **Download** link of the root CA certificate file that corresponds to the type of data endpoint and cipher suite you're using. In this tutorial, choose **Download** to the right of **RSA 2048 bit key: Amazon Root CA 1** and download the **RSA 2048 bit key: Amazon Root CA 1** certificate file.

> **Important**
>
> You must save the certificate files before you leave this page. After you leave this page in the console, you will no longer have access to the certificate files.
> If you forgot to download the certificate files that you created in this step, you must exit this console screen, go to the list of things in the console, delete the thing object you created, and then restart this procedure from the beginning.

3. Choose **Done**.

After you complete this procedure, you should see the new thing object in your list of things.

Configure your device

This section describes how to configure your device to connect to AWS IoT Core. If you'd like to get started with AWS IoT Core but don't have a device yet, you can create a virtual device by using Amazon EC2 or you can use your Windows PC or Mac as an IoT device.

Select the best device option for you to try AWS IoT Core. Of course, you can try all of them, but try only one at a time. If you're not sure which device option is best for you, read about how to choose which device option is the best, and then return to this page.

Device options

- Create a virtual device with Amazon EC2
- Use your Windows or Linux PC or Mac as an AWS IoT device
- Connect a Raspberry Pi or other device

Create a virtual device with Amazon EC2

In this tutorial, you'll create an Amazon EC2 instance to serve as your virtual device in the cloud.

To complete this tutorial, you need an AWS account. If you don't have one, complete the steps described in Set up AWS account before you continue.

In this tutorial, you'll:

- Set up an Amazon EC2 instance
- Install Git, Node.js and configure the AWS CLI
- Create AWS IoT resources for your virtual device
- Install the AWS IoT Device SDK for JavaScript
- Run the sample application
- View messages from the sample app in the AWS IoT console

Set up an Amazon EC2 instance

The following steps show you how to create an Amazon EC2 instance that will act as your virtual device in place of a physical device.

If this is the first time you've created an Amazon EC2 instance, you might find the instructions in Get started with Amazon EC2Linux instances to be more helpful.

To launch an instance

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. From the console menu on the left, expand **Instances** section and choose **Instances**. From the **Instances** dashboard, choose **Launch instances** on the right to display a list of basic configurations.
3. In the **Name and tags** section, enter a name for the instance and optionally add tags.
4. In the **Application and OS Images (Amazon Machine Image)** section, choose an AMI template for your instance, such as *Amazon Linux 2 AMI (HVM)*. Notice that this AMI is marked "Free tier eligible."
5. In the **Instance type** section, you can select the hardware configuration of your instance. Select the `t2.micro` type, which is selected by default. Notice that this instance type is eligible for the free tier.
6. In the **Key pair (login)** section, choose a key pair name from the drop-down list or choose **Create a new key pair** to create a new one. When creating a new key pair, make sure you download the private key file and save it in a safe place, because this is your only chance to

download and save it. You'll need to provide the name of your key pair when you launch an instance and the corresponding private key each time you connect to the instance.

> **Warning**
>
> Don't choose the **Proceed without a key pair** option. If you launch your instance without a key pair, then you can't connect to it.

7. In the **Network settings** section and the **Configure storage** section, you can keep the default settings. When you are ready, choose **Launch instances**.
8. A confirmation page lets you know that your instance is launching. Choose **View Instances** to close the confirmation page and return to the console.
9. On the **Instances** screen, you can view the status of the launch. It takes a short time for an instance to launch. When you launch an instance, its initial state is `pending`. After the instance starts, its state changes to `running` and it receives a public DNS name. (If the **Public DNS (IPv4)** column is hidden, choose **Show/Hide Columns** (the gear-shaped icon) in the top right corner of the page and then select **Public DNS (IPv4)**.)
10. It can take a few minutes for the instance to be ready so that you can connect to it. Check that your instance has passed its status checks; you can view this information in the **Status Checks** column.

    After your new instance has passed its status checks, continue to the next procedure and connect to it.

To connect to your instance

You can connect to an instance using the browser-based client by selecting the instance from the Amazon EC2 console and choosing to connect using Amazon EC2 Instance Connect. Instance Connect handles the permissions and provides a successful connection.

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. On the left menu, choose **Instances**.
3. Select the instance and choose **Connect**.
4. Choose **Amazon EC2 Instance Connect** , **Connect**.

You should now have an **Amazon EC2 Instance Connect** window that is logged into your new Amazon EC2 instance.

Install Git, Node.js and configure the AWS CLI

In this section, you'll install Git and Node.js on your Linux instance.

To install Git

1. In your **Amazon EC2 Instance Connect** window, update your instance by using the following command.

   ```
   sudo yum update -y
   ```

2. In your **Amazon EC2 Instance Connect** window, install Git by using the following command.

   ```
   sudo yum install git -y
   ```

3. To check if Git has been installed and the current version of Git, run the following command:

   ```
   git --version
   ```

To install Node.js

1. In your **Amazon EC2 Instance Connect** window, install node version manager (nvm) by using the following command.

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash
   ```

   We will use nvm to install Node.js because nvm can install multiple versions of Node.js and allow you to switch between them.

2. In your **Amazon EC2 Instance Connect** window, activate nvm by using this command.

   ```
   . ~/.nvm/nvm.sh
   ```

3. In your **Amazon EC2 Instance Connect** window, use nvm to install the latest version of Node.js by using this command.

   ```
   nvm install 16
   ```

> **Note**
>
> This installs the latest LTS release of Node.js.

Installing Node.js also installs the Node Package Manager (npm) so you can install additional modules as needed.

4. In your **Amazon EC2 Instance Connect** window, test that Node.js is installed and running correctly by using this command.

```
node -e "console.log('Running Node.js ' + process.version)"
```

This tutorial requires Node v10.0 or later. For more information, see Tutorial: Setting Up Node.js on an Amazon EC2 Instance.

To configure AWS CLI

Your Amazon EC2 instance comes preloaded with the AWS CLI. However, you must complete your AWS CLI profile. For more information on how to configure your CLI, see Configuring the AWS CLI.

1. The following example shows sample values. Replace them with your own values. You can find these values in your AWS console in your account info under **Security credentials**.

   In your **Amazon EC2 Instance Connect** window, enter this command:

```
aws configure
```

   Then enter the values from your account at the prompts displayed.

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

2. You can test your AWS CLI configuration with this command:

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

If your AWS CLI is configured correctly, the command should return an endpoint address from your AWS account.

Create AWS IoT resources for your virtual device

This section describes how to use the AWS CLI to create the thing object and its certificate files directly on the virtual device. This is done directly on the device to avoid the potential complication that might arise from copying them to the device from another computer. In this section, you will create the following resources for your virtual device:

- A thing object to represent your virtual device in AWS IoT.
- A certificate to authenticate your virtual device.
- A policy document to authorize your virtual device to connect to AWS IoT, and to publish, receive, and subscribe to messages.

To create an AWS IoT thing object in your Linux instance

Devices connected to AWS IoT are represented by *thing objects* in the AWS IoT registry. A *thing object* represents a specific device or logical entity. In this case, your *thing object* will represent your virtual device, this Amazon EC2 instance.

1. In your **Amazon EC2 Instance Connect** window, run the following command to create your thing object.

   ```
   aws iot create-thing --thing-name "MyIotThing"
   ```

2. The JSON response should look like this:

   ```
   {
       "thingArn": "arn:aws:iot:your-region:your-aws-account:thing/MyIotThing",
       "thingName": "MyIotThing",
       "thingId": "6cf922a8-d8ea-4136-f3401EXAMPLE"
   }
   ```

To create and attach AWS IoT keys and certificates in your Linux instance

The create-keys-and-certificate command creates client certificates signed by the Amazon Root certificate authority. This certificate is used to authenticate the identity of your virtual device.

1. In your **Amazon EC2 Instance Connect** window, create a directory to store your certificate and key files.

```
mkdir ~/certs
```

2. In your **Amazon EC2 Instance Connect** window, download a copy of the Amazon certificate authority (CA) certificate by using this command.

```
curl -o ~/certs/Amazon-root-CA-1.pem \
    https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

3. In your **Amazon EC2 Instance Connect** window, run the following command to create your private key, public key, and X.509 certificate files. This command also registers and activates the certificate with AWS IoT.

```
aws iot create-keys-and-certificate \
    --set-as-active \
    --certificate-pem-outfile "~/certs/device.pem.crt" \
    --public-key-outfile "~/certs/public.pem.key" \
    --private-key-outfile "~/certs/private.pem.key"
```

The response looks like the following. Save the `certificateArn` so that you can use it in subsequent commands. You'll need it to attach your certificate to your thing and to attach the policy to the certificate in a later steps.


```
{
    "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/9894ba17925e663f1d29c23af4582b8e3b7619c31f3fbd93adcb51ae54b83dc2",
    "certificateId":
 "9894ba17925e663f1d29c23af4582b8e3b7619c31f3fbd93adcb51ae54b83dc2",
    "certificatePem": "
-----BEGIN CERTIFICATE-----
MIICiTCCEXAMPLE6m7oRw0uXOjANBgkqhkiG9w0BAQUFADCBiDELMAkGA1UEBhMC
VVMxCzAJBgNVBAgEXAMPLEAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6
b24xFDASBgNVBAsTC0lBTSEXAMPLE2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMxHzAd
BgkqhkiG9w0BCQEWEG5vb25lQGFtYEXAMPLEb20wHhcNMTEwNDI1MjA0NTIxWhcN
```

```
MTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBhMCEXAMPLEJBgNVBAgTAldBMRAwDgYD
VQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDAEXAMPLEsTC0lBTSBDb25z
b2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEXAMPLE25lQGFt
YXpvbi5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMaK0dn+aEXAMPLE
EXAMPLEfEvySWtC2XADZ4nB+BLYgVIk60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9T
rDHudUZEXAMPLELG5M43q7Wgc/MbQITxOUSQv7c7ugFFDzQGBzZswY6786m86gpE
Ibb3OhjZnzcvQAEXAMPLEWIMm2nrAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4
nUhVVxYUntneD9+h8Mg9qEXAMPLEyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0Fkb
FFBjvSfpJIlJ00zbhNYS5f6GuoEDEXAMPLEBHjJnyp378OD8uTs7fLvjx79LjSTb
NYiytVbZPQUQ5Yaxu2jXnimvw3rrszlaEXAMPLE=
-----END CERTIFICATE-----\n",
    "keyPair": {
        "PublicKey": "-----BEGIN PUBLIC
 KEY-----\nMIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h
\nMMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/
gHr99VEEXAMPLE5VF13\n59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa
\nhJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\nFQIDAQAB\n-----
END PUBLIC KEY-----\n",
        "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\nkey omitted for security
 reasons\n-----END RSA PRIVATE KEY-----\n"
    }
}
```

4. In your **Amazon EC2 Instance Connect** window, attach your thing object to the certificate you just created by using the following command and the *`certificateArn`* in the response from the previous command.

```
aws iot attach-thing-principal \
    --thing-name "MyIotThing" \
    --principal "certificateArn"
```

   If successful, this command does not display any output.

To create and attach a policy

1. In your **Amazon EC2 Instance Connect** window, create the policy file by copying and pasting this policy document to a file named **`~/policy.json`**.

   If you don't have a favorite Linux editor, you can open **nano**, by using this command.

```
nano ~/policy.json
```

Paste the policy document for `policy.json` into it. Enter ctrl-x to exit the **nano** editor and save the file.

Contents of the policy document for `policy.json`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Subscribe",
                "iot:Receive",
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

2. In your **Amazon EC2 Instance Connect** window, create your policy by using the following command.

```
aws iot create-policy \
    --policy-name "MyIotThingPolicy" \
    --policy-document "file://~/policy.json"
```

Output:

```
{
    "policyName": "MyIotThingPolicy",
    "policyArn": "arn:aws:iot:us-east-1:your-aws-account:policy/
MyIotThingPolicy",
    "policyDocument": "{
```

```
        \"Version\": \"2012-10-17\",
        \"Statement\": [
            {
                \"Effect\": \"Allow\",
                \"Action\": [
                    \"iot:Publish\",
                    \"iot:Receive\",
                    \"iot:Subscribe\",
                    \"iot:Connect\"
                ],
                \"Resource\": [
                    \"*\"
                ]
        }
        ]
    }",
    "policyVersionId": "1"
}
```

3. In your **Amazon EC2 Instance Connect** window, attach the policy to your virtual device's certificate by using the following command.

```
aws iot attach-policy \
    --policy-name "MyIotThingPolicy" \
    --target "certificateArn"
```

   If successful, this command does not display any output.

Install the AWS IoT Device SDK for JavaScript

In this section, you'll install the AWS IoT Device SDK for JavaScript, which contains the code that applications can use to communicate with AWS IoT and the sample programs. For more information, see the AWS IoT Device SDK for JavaScript GitHub repository.

To install the AWS IoT Device SDK for JavaScript on your Linux instance

1. In your **Amazon EC2 Instance Connect** window, clone the AWS IoT Device SDK for JavaScript repository into the `aws-iot-device-sdk-js-v2` directory of your home directory by using this command.

```
cd ~
```

```
git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
```

2. Navigate to the `aws-iot-device-sdk-js-v2` directory that you created in the preceding step.

```
cd aws-iot-device-sdk-js-v2
```

3. Use npm to install the SDK.

```
npm install
```

Run the sample application

The commands in the next sections assume that your key and certificate files are stored on your virtual device as shown in this table.

Certificate file names

File	File path
Private key	`~/certs/private.pem.key`
Device certificate	`~/certs/device.pem.crt`
Root CA certificate	`~/certs/Amazon-root-CA-1.pem`

In this section, you'll install and run the `pub-sub.js` sample app found in the `aws-iot-device-sdk-js-v2/samples/node` directory of the AWS IoT Device SDK for JavaScript. This app shows how a device, your Amazon EC2 instance, uses the MQTT library to publish and subscribe to MQTT messages. The `pub-sub.js` sample app subscribes to a topic, `topic_1`, publishes 10 messages to that topic and displays the messages as they're received from the message broker.

To install and run the sample app

1. In your **Amazon EC2 Instance Connect** window, navigate to the `aws-iot-device-sdk-js-v2/samples/node/pub_sub` directory that the SDK created and install the sample app by using these commands.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
```

```
npm install
```

2. In your **Amazon EC2 Instance Connect** window, get *your-iot-endpoint* from AWS IoT by using this command.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

3. In your **Amazon EC2 Instance Connect** window, insert *your-iot-endpoint* as indicated and run this command.

```
node dist/index.js --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/
certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

The sample app:

1. Connects to AWS IoT Core for your account.
2. Subscribes to the message topic, **topic_1**, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, **topic_1**.
4. Displays output similar to the following:

```
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":1}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":2}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":3}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":4}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":5}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":6}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":7}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":8}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":9}
Publish received. topic:"topic_1" dup:false qos:1 retain:false
{"message":"Hello world!","sequence":10}
```

If you're having trouble running the sample app, review the section called “Troubleshoot problems with the sample application”.

You can also add the `--verbosity debug` parameter to the command line so the sample app displays detailed messages about what it’s doing. That information might provide you the help you need to correct the problem.

View messages from the sample app in the AWS IoT console

You can see the sample app's messages as they pass through the message broker by using the **MQTT test client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review View MQTT messages with the AWS IoT MQTT client. This helps you learn how to use the **MQTT test client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT test client** in the **AWS IoT console**.
3. In **Subscribe to a topic**, Subscribe to the topic, **topic_1**.
4. In your **Amazon EC2 Instance Connect** window, run the sample app again and watch the messages in the **MQTT test client** in the **AWS IoT console**.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
node dist/index.js --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/
certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

For more information about MQTT and how AWS IoT Core supports the protocol, see MQTT.

Use your Windows or Linux PC or Mac as an AWS IoT device

In this tutorial, you'll configure a personal computer for use with AWS IoT. These instructions support Windows and Linux PCs and Macs. To accomplish this, you need to install some software on your computer. If you don't want to install software on your computer, you might try Create a virtual device with Amazon EC2, which installs all software on a virtual machine.

In this tutorial, you'll:

- Set up your personal computer
- Install Git, Python, and the AWS IoT Device SDK for Python

- Set up the policy and run the sample application
- View messages from the sample app in the AWS IoT console
- Run the Shared Subscription example in Python

Set up your personal computer

To complete this tutorial, you need a Windows or Linux PC or a Mac with a connection to the internet.

Before you continue to the next step, make sure you can open a command line window on your computer. Use **cmd.exe** on a Windows PC. On a Linux PC or a Mac, use **Terminal**.

Install Git, Python, and the AWS IoT Device SDK for Python

In this section, you'll install Python, and the AWS IoT Device SDK for Python on your computer.

Install the latest version of Git and Python

This procedure explains how to install the latest version of Git and Python on your personal computer.

To download and install Git and Python on your computer

1. Check to see if you have Git installed on your computer. Enter this command in the command line.

   ```
   git --version
   ```

   If the command displays the Git version, Git is installed and you can continue to the next step.

   If the command displays an error, open https://git-scm.com/download and install Git for your computer.

2. Check to see if you have already installed Python. Enter the command in the command line.

   ```
   python -V
   ```

> **Note**
>
> If this command gives an error: `Python was not found`, it might be because your operating system calls the Python v3.x executable as `Python3`. In that case, replace all instances of `python` with `python3` and continue the remainder of this tutorial.

If the command displays the Python version, Python is already installed. This tutorial requires Python v3.7 or later.

3. If Python is installed, you can skip the rest of the steps in this section. If not, continue.
4. Open https://www.python.org/downloads/ and download the installer for your computer.
5. If the download didn't automatically start to install, run the downloaded program to install Python.
6. Verify the installation of Python.

```
python -V
```

Confirm that the command displays the Python version. If the Python version isn't displayed, try downloading and installing Python again.

Install the AWS IoT Device SDK for Python

To install the AWS IoT Device SDK for Python on your computer

1. Install v2 of the AWS IoT Device SDK for Python.

```
python3 -m pip install awsiotsdk
```

2. Clone the AWS IoT Device SDK for Python repository into the aws-iot-device-sdk-python-v2 directory of your home directory. This procedure refers to the base directory for the files you're installing as *home*.

   The actual location of the *home* directory depends on your operating system.

   Linux/macOS

   In macOS and Linux, the *home* directory is ~.

```
cd ~
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

Windows

In Windows, you can find the *home* directory path by running this command in the cmd window.

```
echo %USERPROFILE%
cd %USERPROFILE%
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

> **Note**
>
> If you're using Windows PowerShell as opposed to **cmd.exe**, then use the following command.
>
> ```
> echo $home
> ```

For more information, see the AWS IoT Device SDK for Python GitHub repository.

Prepare to run the sample applications

To prepare your system to run the sample application

- Create the `certs` directory. Into the `certs` directory, copy the private key, device certificate, and root CA certificate files you saved when you created and registered the thing object in the section called "Create AWS IoT resources". The file names of each file in the destination directory should match those in the table.

  The commands in the next section assume that your key and certificate files are stored on your device as shown in this table.

Linux/macOS

Run this command to create the `certs` subdirectory that you'll use when you run the sample applications.

```
mkdir ~/certs
```

Into the new subdirectory, copy the files to the destination file paths shown in the following table.

Certificate file names

File	File path
Private key	`~/certs/private.pem.key`
Device certificate	`~/certs/device.pem.crt`
Root CA certificate	`~/certs/Amazon-root-CA-1.pem`

Run this command to list the files in the `certs` directory and compare them to those listed in the table.

```
ls -l ~/certs
```

Windows

Run this command to create the `certs` subdirectory that you'll use when you run the sample applications.

```
mkdir %USERPROFILE%\certs
```

Into the new subdirectory, copy the files to the destination file paths shown in the following table.

Certificate file names

File	File path
Private key	`%USERPROFILE%\certs\private.pem.key`
Device certificate	`%USERPROFILE%\certs\device.pem.crt`
Root CA certificate	`%USERPROFILE%\certs\Amazon-root-CA-1.pem`

Run this command to list the files in the `certs` directory and compare them to those listed in the table.

```
dir %USERPROFILE%\certs
```

Set up the policy and run the sample application

In this section, you'll set up your policy and run the `pubsub.py` sample script found in the `aws-iot-device-sdk-python-v2/samples` directory of the AWS IoT Device SDK for Python. This script shows how your device uses the MQTT library to publish and subscribe to MQTT messages.

The `pubsub.py` sample app subscribes to a topic, `test/topic`, publishes 10 messages to that topic, and displays the messages as they're received from the message broker.

To run the `pubsub.py` sample script, you need the following information:

Application parameter values

Parameter	Where to find the value
`your-iot-endpoint`	1. In the AWS IoT console, in the left menu, choose **Settings**. 2. On the **Settings** page, your endpoint is displayed in the **Device data endpoint** section.

The *your-iot-endpoint* value has a format of: *endpoint_id*-ats.iot.*region*.amazonaws.com, for example, a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com.

Before running the script, make sure your thing's policy provides permissions for the sample script to connect, subscribe, publish, and receive.

To find and review the policy document for a thing resource

1. In the AWS IoT console, in the **Things** list, find the thing resource that represents your device.
2. Choose the **Name** link of the thing resource that represents your device to open the **Thing details** page.
3. In the **Thing details** page, in the **Certificates** tab, choose the certificate that is attached to the thing resource. There should only be one certificate in the list. If there is more than one, choose the certificate whose files are installed on your device and that will be used to connect to AWS IoT Core.

   In the **Certificate** details page, in the **Policies** tab, choose the policy that's attached to the certificate. There should only be one. If there is more than one, repeat the next step for each to make sure that at least one policy grants the required access.
4. In the **Policy** overview page, find the JSON editor and choose **Edit policy document** to review and edit the policy document as required.
5. The policy JSON is displayed in the following example. In the "Resource" element, replace *region:account* with your AWS Region and AWS account in each of the Resource values.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/test/topic"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/test/topic"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:client/test-*"
            ]
        }
    ]
}
```

Linux/macOS

To run the sample script on Linux/macOS

1. In your command line window, navigate to the `~/aws-iot-device-sdk-python-v2/samples/node/pub_sub` directory that the SDK created by using these commands.

```
cd ~/aws-iot-device-sdk-python-v2/samples
```

2. In your command line window, replace *`your-iot-endpoint`* as indicated and run this command.

```
python3 pubsub.py --endpoint your-iot-endpoint --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key
```

Windows

To run the sample app on a Windows PC

1. In your command line window, navigate to the `%USERPROFILE%\aws-iot-device-sdk-python-v2\samples` directory that the SDK created and install the sample app by using these commands.

```
cd %USERPROFILE%\aws-iot-device-sdk-python-v2\samples
```

2. In your command line window, replace *your-iot-endpoint* as indicated and run this command.

```
python3 pubsub.py --endpoint your-iot-endpoint --ca_file %USERPROFILE%
\certs\Amazon-root-CA-1.pem --cert %USERPROFILE%\certs\device.pem.crt --key
 %USERPROFILE%\certs\private.pem.key
```

The sample script:

1. Connects to the AWS IoT Core for your account.
2. Subscribes to the message topic, **test/topic**, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, **test/topic**.
4. Displays output similar to the following:

```
Connected!
Subscribing to topic 'test/topic'...
Subscribed with QoS.AT_LEAST_ONCE
Sending 10 message(s)
Publishing message to topic 'test/topic': Hello World! [1]
Received message from topic 'test/topic': b'"Hello World! [1]"'
Publishing message to topic 'test/topic': Hello World! [2]
Received message from topic 'test/topic': b'"Hello World! [2]"'
Publishing message to topic 'test/topic': Hello World! [3]
Received message from topic 'test/topic': b'"Hello World! [3]"'
Publishing message to topic 'test/topic': Hello World! [4]
Received message from topic 'test/topic': b'"Hello World! [4]"'
Publishing message to topic 'test/topic': Hello World! [5]
Received message from topic 'test/topic': b'"Hello World! [5]"'
Publishing message to topic 'test/topic': Hello World! [6]
```

```
Received message from topic 'test/topic': b'"Hello World! [6]"'
Publishing message to topic 'test/topic': Hello World! [7]
Received message from topic 'test/topic': b'"Hello World! [7]"'
Publishing message to topic 'test/topic': Hello World! [8]
Received message from topic 'test/topic': b'"Hello World! [8]"'
Publishing message to topic 'test/topic': Hello World! [9]
Received message from topic 'test/topic': b'"Hello World! [9]"'
Publishing message to topic 'test/topic': Hello World! [10]
Received message from topic 'test/topic': b'"Hello World! [10]"'
10 message(s) received.
Disconnecting...
Disconnected!
```

If you're having trouble running the sample app, review the section called “Troubleshoot problems with the sample application”.

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it’s doing. That information might help you correct the problem.

View messages from the sample app in the AWS IoT console

You can see the sample app's messages as they pass through the message broker by using the **MQTT test client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review View MQTT messages with the AWS IoT MQTT client. This helps you learn how to use the **MQTT test client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT test client** in the **AWS IoT console**.
3. In **Subscribe to a topic**, subscribe to the topic, **test/topic**.
4. In your command line window, run the sample app again and watch the messages in the **MQTT client** in the **AWS IoT console**.

   Linux/macOS

   ```
   cd ~/aws-iot-device-sdk-python-v2/samples
   ```

```
python3 pubsub.py --topic test/topic --ca_file ~/certs/Amazon-root-CA-1.pem --
cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-
endpoint
```

Windows

```
cd %USERPROFILE%\aws-iot-device-sdk-python-v2\samples
python3 pubsub.py --topic test/topic --ca_file %USERPROFILE%\certs\Amazon-root-
CA-1.pem --cert %USERPROFILE%\certs\device.pem.crt --key %USERPROFILE%\certs
\private.pem.key --endpoint your-iot-endpoint
```

For more information about MQTT and how AWS IoT Core supports the protocol, see MQTT.

Run the Shared Subscription example in Python

AWS IoT Core supports Shared Subscriptions for both MQTT 3 and MQTT 5. Shared Subscriptions allow multiple clients to share a subscription to a topic and only one client will receive messages published to that topic using a random distribution. To use Shared Subscriptions, clients subscribe to a Shared Subscription's topic filter: `$share/{ShareName}/{TopicFilter}`.

To set up the policy and run the Shared Subscription example

1. To run the Shared Subscription example, you must set up your thing's policy as documented in MQTT 5 Shared Subscription.
2. To run the Shared Subscription example, run the following commands.

   Linux/macOS

   To run the sample script on Linux/macOS

   1. In your command line window, navigate to the `~/aws-iot-device-sdk-python-v2/samples` directory that the SDK created by using these commands.

      ```
      cd ~/aws-iot-device-sdk-python-v2/samples
      ```

   2. In your command line window, replace *`your-iot-endpoint`* as indicated and run this command.

```
python3 mqtt5_shared_subscription.py --endpoint your-iot-endpoint --ca_file
 ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/
private.pem.key --group_identifier consumer
```

Windows

To run the sample app on a Windows PC

1. In your command line window, navigate to the `%USERPROFILE%\aws-iot-device-sdk-python-v2\samples` directory that the SDK created and install the sample app by using these commands.

   ```
   cd %USERPROFILE%\aws-iot-device-sdk-python-v2\samples
   ```

2. In your command line window, replace *`your-iot-endpoint`* as indicated and run this command.

   ```
   python3 mqtt5_shared_subscription.py --endpoint your-iot-endpoint --ca_file
    %USERPROFILE%\certs\Amazon-root-CA-1.pem --cert %USERPROFILE%\certs
   \device.pem.crt --key %USERPROFILE%\certs\private.pem.key --group_identifier
    consumer
   ```

> **Note**
>
> You can optionally specify a group identifier based on your needs when you run the sample (e.g., `--group_identifier consumer`). If you don't specify one, `python-sample` is the default group identifier.

3. The output in your command line can look like the following:

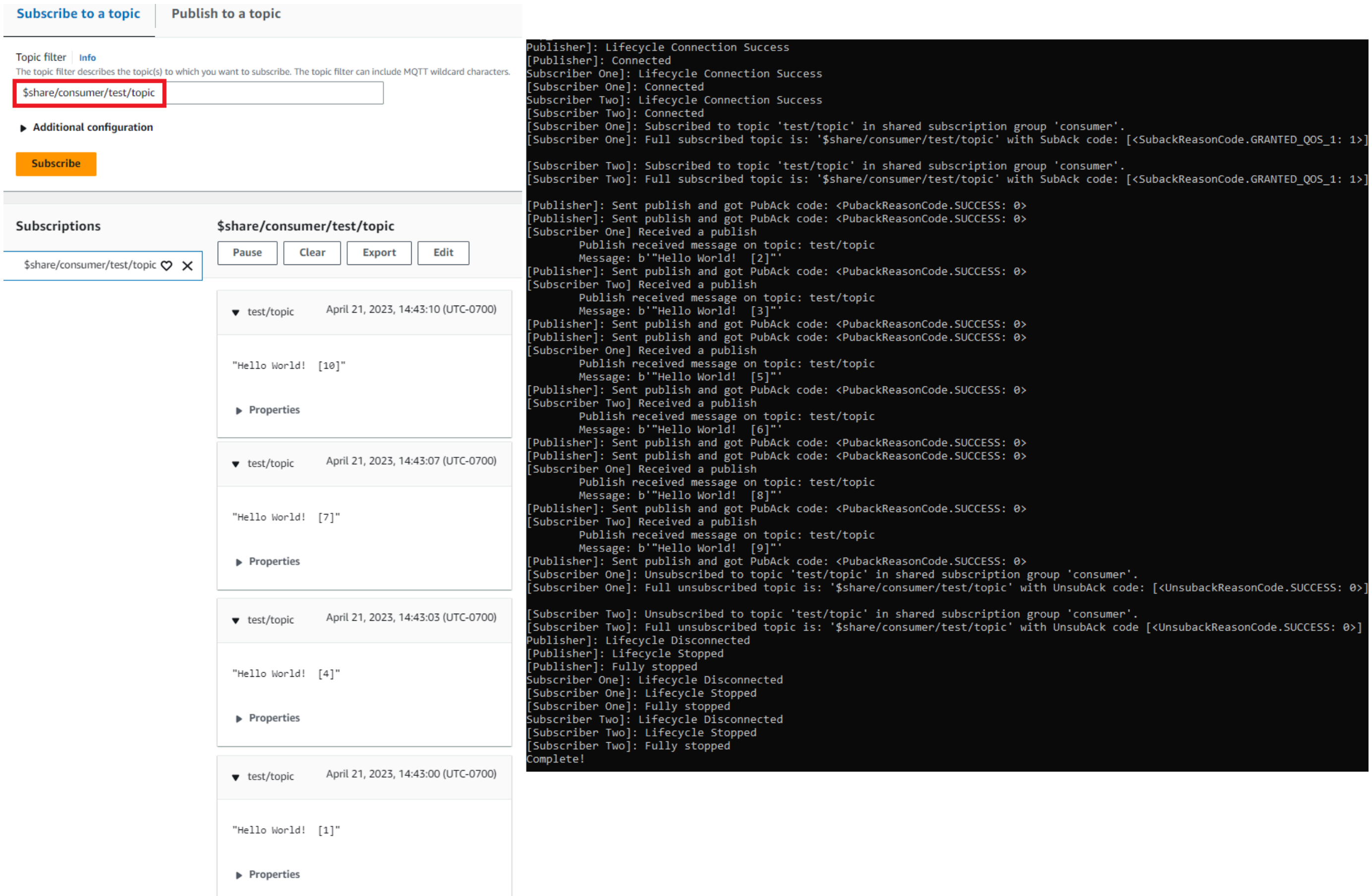
```
Publisher]: Lifecycle Connection Success
[Publisher]: Connected
Subscriber One]: Lifecycle Connection Success
[Subscriber One]: Connected
Subscriber Two]: Lifecycle Connection Success
[Subscriber Two]: Connected
[Subscriber One]: Subscribed to topic 'test/topic' in shared subscription group
 'consumer'.
```

```
[Subscriber One]: Full subscribed topic is: '$share/consumer/test/topic' with
 SubAck code: [<SubackReasonCode.GRANTED_QOS_1: 1>]
[Subscriber Two]: Subscribed to topic 'test/topic' in shared subscription group
 'consumer'.
[Subscriber Two]: Full subscribed topic is: '$share/consumer/test/topic' with
 SubAck code: [<SubackReasonCode.GRANTED_QOS_1: 1>]
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber Two] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [1]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber One] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [2]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber Two] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [3]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber One] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [4]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber Two] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [5]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber One] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [6]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber Two] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [7]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber One] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [8]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber Two] Received a publish
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [9]"'
[Publisher]: Sent publish and got PubAck code: <PubackReasonCode.SUCCESS: 0>
[Subscriber One] Received a publish
```

```
        Publish received message on topic: test/topic
        Message: b'"Hello World!  [10]"'
[Subscriber One]: Unsubscribed to topic 'test/topic' in shared subscription group
 'consumer'.
[Subscriber One]: Full unsubscribed topic is: '$share/consumer/test/topic' with
 UnsubAck code: [<UnsubackReasonCode.SUCCESS: 0>]
[Subscriber Two]: Unsubscribed to topic 'test/topic' in shared subscription group
 'consumer'.
[Subscriber Two]: Full unsubscribed topic is: '$share/consumer/test/topic' with
 UnsubAck code [<UnsubackReasonCode.SUCCESS: 0>]
Publisher]: Lifecycle Disconnected
[Publisher]: Lifecycle Stopped
[Publisher]: Fully stopped
Subscriber One]: Lifecycle Disconnected
[Subscriber One]: Lifecycle Stopped
[Subscriber One]: Fully stopped
Subscriber Two]: Lifecycle Disconnected
[Subscriber Two]: Lifecycle Stopped
[Subscriber Two]: Fully stopped
Complete!
```

4. Open **MQTT test client** in the **AWS IoT console**. In **Subscribe to a topic**, subscribe to the Shared Subscription's topic such as: `$share/consumer/test/topic`. You can specify a group identifier based on your needs when you run the sample (e.g., `--group_identifier consumer`). If you don't specify a group identifier, the default value is `python-sample`. For more information, see MQTT 5 Shared Subscription Python example and Shared Subscriptions from *AWS IoT Core Developer Guide*.

   In your command line window, run the sample app again and watch the distribution of messages in your **MQTT test client** of the **AWS IoT console** and the command line.



Connect a Raspberry Pi or other device

In this section, we'll configure a Raspberry Pi for use with AWS IoT. If you have another device that you'd like to connect, the instructions for the Raspberry Pi include references that can help you adapt these instructions to your device.

This normally takes about 20 minutes, but it can take longer if you have many system software upgrades to install.

In this tutorial, you'll:

- Set up your device
- Install the required tools and libraries for the AWS IoT Device SDK
- Install AWS IoT Device SDK
- Install and run the sample app
- View messages from the sample app in the AWS IoT console

> **Important**
>
> Adapting these instructions to other devices and operating systems can be challenging. You'll need to understand your device well enough to be able to interpret these instructions and apply them to your device.
> If you encounter difficulties while configuring your device for AWS IoT, you might try one of the other device options as an alternative, such as Create a virtual device with Amazon EC2 or Use your Windows or Linux PC or Mac as an AWS IoT device.

Set up your device

The goal of this step is to collect what you'll need to configure your device so that it can start the operating system (OS), connect to the internet, and allow you to interact with it at a command line interface.

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, complete the steps described in Set up AWS account before you continue.
- A Raspberry Pi 3 Model B or more recent model. This might work on earlier versions of the Raspberry Pi, but they have not been tested.
- Raspberry Pi OS (32-bit) or later. We recommend using the latest version of the Raspberry Pi OS. Earlier versions of the OS might work, but they have not been tested.

  To run this example, you do not need to install the desktop with the graphical user interface (GUI); however, if you're new to the Raspberry Pi and your Raspberry Pi hardware supports it, using the desktop with the GUI might be easier.
- An Ethernet or WiFi connection.
- Keyboard, mouse, monitor, cables, power supplies, and other hardware required by your device.

> **Important**
>
> Before you continue to the next step, your device must have its operating system installed, configured, and running. The device must be connected to the internet and you will need to be able to access the device by using its command line interface. Command line access

can be through a directly-connected keyboard, mouse, and monitor, or by using an SSH terminal remote interface.

If you are running an operating system on your Raspberry Pi that has a graphical user interface (GUI), open a terminal window on the device and perform the following instructions in that window. Otherwise, if you are connecting to your device by using a remote terminal, such as PuTTY, open a remote terminal to your device and use that.

Install the required tools and libraries for the AWS IoT Device SDK

Before you install the AWS IoT Device SDK and sample code, make sure your system is up to date and has the required tools and libraries to install the SDKs.

1. **Update the operating system and install required libraries**

   Before you install an AWS IoT Device SDK, run these commands in a terminal window on your device to update the operating system and install the required libraries.

   ```
   sudo apt-get update
   ```

   ```
   sudo apt-get upgrade
   ```

   ```
   sudo apt-get install cmake
   ```

   ```
   sudo apt-get install libssl-dev
   ```

2. **Install Git**

   If your device's operating system doesn't come with Git installed, you must install it to install the AWS IoT Device SDK for JavaScript.

   a. Test to see if Git is already installed by running this command.

      ```
      git --version
      ```

   b. If the previous command returns the Git version, Git is already installed and you can skip to Step 3.

c. If an error is displayed when you run the **git** command, install Git by running this command.

```
sudo apt-get install git
```

d. Test again to see if Git is installed by running this command.

```
git --version
```

e. If Git is installed, continue to the next section. If not, troubleshoot and correct the error before continuing. You need Git to install the AWS IoT Device SDK for JavaScript.

Install AWS IoT Device SDK

Install the AWS IoT Device SDK.

Python

In this section, you'll install Python, its development tools, and the AWS IoT Device SDK for Python on your device. These instructions are for a Raspberry Pi running the latest Raspberry Pi OS. If you have another device or are using another operating system, you might need to adapt these instructions for your device.

1. **Install Python and its development tools**

   The AWS IoT Device SDK for Python requires Python v3.5 or later to be installed on your Raspberry Pi.

   In a terminal window to your device, run these commands.

   1. Run this command to determine the version of Python installed on your device.

      ```
      python3 --version
      ```

      If Python is installed, it will display its version.
   2. If the version displayed is `Python 3.5` or greater, you can skip to Step 2.
   3. If the version displayed is less than `Python 3.5`, you can install the correct version by running this command.

```
sudo apt install python3
```

4. Run this command to confirm that the correct version of Python is now installed.

```
python3 --version
```

2. **Test for pip3**

   In a terminal window to your device, run these commands.

   1. Run this command to see if **pip3** is installed.

   ```
   pip3 --version
   ```

   2. If the command returns a version number, **pip3** is installed and you can skip to Step 3.
   3. If the previous command returns an error, run this command to install **pip3**.

   ```
   sudo apt install python3-pip
   ```

   4. Run this command to see if **pip3** is installed.

   ```
   pip3 --version
   ```

3. **Install the current AWS IoT Device SDK for Python**

   Install the AWS IoT Device SDK for Python and download the sample apps to your device.

   On your device, run these commands.

   ```
   cd ~
   python3 -m pip install awsiotsdk
   ```

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
   ```

JavaScript

In this section, you'll install Node.js, the npm package manager, and the AWS IoT Device SDK for JavaScript on your device. These instructions are for a Raspberry Pi running the Raspberry Pi

OS. If you have another device or are using another operating system, you might need to adapt these instructions for your device.

1. **Install the latest version of Node.js**

   The AWS IoT Device SDK for JavaScript requires Node.js and the npm package manager to be installed on your Raspberry Pi.

   a. Download the latest version of the Node repository by entering this command.

      ```
      cd ~
      curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
      ```

   b. Install Node and npm.

      ```
      sudo apt-get install -y nodejs
      ```

   c. Verify the installation of Node.

      ```
      node -v
      ```

      Confirm that the command displays the Node version. This tutorial requires Node v10.0 or later. If the Node version isn't displayed, try downloading the Node repository again.

   d. Verify the installation of npm.

      ```
      npm -v
      ```

      Confirm that the command displays the npm version. If the npm version isn't displayed, try installing Node and npm again.

   e. Restart the device.

      ```
      sudo shutdown -r 0
      ```

      Continue after the device restarts.

2. **Install the AWS IoT Device SDK for JavaScript**

   Install the AWS IoT Device SDK for JavaScript on your Raspberry Pi.

a. Clone the AWS IoT Device SDK for JavaScript repository into the `aws-iot-device-sdk-js-v2` directory of your *home* directory. On the Raspberry Pi, the *home* directory is `~/`, which is used as the *home* directory in the following commands. If your device uses a different path for the *home* directory, you must replace `~/` with the correct path for your device in the following commands.

   These commands create the `~/aws-iot-device-sdk-js-v2` directory and copy the SDK code into it.

```
cd ~
git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
```

b. Change to the `aws-iot-device-sdk-js-v2` directory that you created in the preceding step and run `npm install` to install the SDK. The command `npm install` will invoke the `aws-crt` library build that can take a few minutes to complete.

```
cd ~/aws-iot-device-sdk-js-v2
npm install
```

Install and run the sample app

In this section, you'll install and run the `pubsub` sample app found in the AWS IoT Device SDK. This app shows how your device uses the MQTT library to publish and subscribe to MQTT messages. The sample app subscribes to a topic, `topic_1`, publishes 10 messages to that topic, and displays the messages as they're received from the message broker.

Install the certificate files

The sample app requires the certificate files that authenticate the device to be installed on the device.

To install the device certificate files for the sample app

1. Create a `certs` subdirectory in your *home* directory by running these commands.

```
cd ~
mkdir certs
```

2. Into the `~/certs` directory, copy the private key, device certificate, and root CA certificate that you created earlier in the section called "Create AWS IoT resources".

   How you copy the certificate files to your device depends on the device and operating system and isn't described here. However, if your device supports a graphical user interface (GUI) and has a web browser, you can perform the procedure described in the section called "Create AWS IoT resources" from your device's web browser to download the resulting files directly to your device.

   The commands in the next section assume that your key and certificate files are stored on the device as shown in this table.

   Certificate file names

File	File path
Root CA certificate	`~/certs/Amazon-root-CA-1.pem`
Device certificate	`~/certs/device.pem.crt`
Private key	`~/certs/private.pem.key`

To run the sample app, you need the following information:

Application parameter values

Parameter	Where to find the value
`your-iot-endpoint`	In the AWS IoT console, choose **All devices**, and then choose **Things**. On the **Settings** page in the AWS IoT menu. Your endpoint is displayed in the **Device data endpoint** section.

The *`your-iot-endpoint`* value has a format of: `endpoint_id-ats.iot.region.amazonaws.com`, for example, `a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com`.

Python

To install and run the sample app

1. Navigate to the sample app directory.

```
cd ~/aws-iot-device-sdk-python-v2/samples
```

2. In the command line window, replace *your-iot-endpoint* as indicated and run this command.

```
python3 pubsub.py --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --
cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-
endpoint
```

3. Observe that the sample app:

   1. Connects to the AWS IoT service for your account.
   2. Subscribes to the message topic, **topic_1**, and displays the messages it receives on that topic.
   3. Publishes 10 messages to the topic, **topic_1**.
   4. Displays output similar to the following:

```
Connecting to a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com with client ID
 'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...
Connected!
Subscribing to topic 'topic_1'...
Subscribed with QoS.AT_LEAST_ONCE
Sending 10 message(s)
Publishing message to topic 'topic_1': Hello World! [1]
Received message from topic 'topic_1': b'Hello World! [1]'
Publishing message to topic 'topic_1': Hello World! [2]
Received message from topic 'topic_1': b'Hello World! [2]'
Publishing message to topic 'topic_1': Hello World! [3]
Received message from topic 'topic_1': b'Hello World! [3]'
Publishing message to topic 'topic_1': Hello World! [4]
Received message from topic 'topic_1': b'Hello World! [4]'
Publishing message to topic 'topic_1': Hello World! [5]
Received message from topic 'topic_1': b'Hello World! [5]'
Publishing message to topic 'topic_1': Hello World! [6]
```

```
Received message from topic 'topic_1': b'Hello World! [6]'
Publishing message to topic 'topic_1': Hello World! [7]
Received message from topic 'topic_1': b'Hello World! [7]'
Publishing message to topic 'topic_1': Hello World! [8]
Received message from topic 'topic_1': b'Hello World! [8]'
Publishing message to topic 'topic_1': Hello World! [9]
Received message from topic 'topic_1': b'Hello World! [9]'
Publishing message to topic 'topic_1': Hello World! [10]
Received message from topic 'topic_1': b'Hello World! [10]'
10 message(s) received.
Disconnecting...
Disconnected!
```

If you're having trouble running the sample app, review the section called "Troubleshoot problems with the sample application".

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it's doing. That information might provide you the help you need to correct the problem.

JavaScript

To install and run the sample app

1. In your command line window, navigate to the `~/aws-iot-device-sdk-js-v2/samples/node/pub_sub` directory that the SDK created and install the sample app by using these commands. The command `npm install` will invoke the `aws-crt` library build that can take a few minutes to complete.

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
npm install
```

2. In the command line window, replace *your-iot-endpoint* as indicated and run this command.

```
node dist/index.js --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

3. Observe that the sample app:

1. Connects to the AWS IoT service for your account.
2. Subscribes to the message topic, **topic_1**, and displays the messages it receives on that topic.
3. Publishes 10 messages to the topic, **topic_1**.
4. Displays output similar to the following:

```
Publish received on topic topic_1
{"message":"Hello world!","sequence":1}
Publish received on topic topic_1
{"message":"Hello world!","sequence":2}
Publish received on topic topic_1
{"message":"Hello world!","sequence":3}
Publish received on topic topic_1
{"message":"Hello world!","sequence":4}
Publish received on topic topic_1
{"message":"Hello world!","sequence":5}
Publish received on topic topic_1
{"message":"Hello world!","sequence":6}
Publish received on topic topic_1
{"message":"Hello world!","sequence":7}
Publish received on topic topic_1
{"message":"Hello world!","sequence":8}
Publish received on topic topic_1
{"message":"Hello world!","sequence":9}
Publish received on topic topic_1
{"message":"Hello world!","sequence":10}
```

If you're having trouble running the sample app, review the section called “Troubleshoot problems with the sample application”.

You can also add the `--verbosity Debug` parameter to the command line so the sample app displays detailed messages about what it’s doing. That information might provide you the help you need to correct the problem.

View messages from the sample app in the AWS IoT console

You can see the sample app's messages as they pass through the message broker by using the **MQTT test client** in the **AWS IoT console**.

To view the MQTT messages published by the sample app

1. Review View MQTT messages with the AWS IoT MQTT client. This helps you learn how to use the **MQTT test client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.
2. Open the **MQTT test client** in the **AWS IoT console**.
3. Subscribe to the topic, **topic_1**.
4. In your command line window, run the sample app again and watch the messages in the **MQTT client** in the **AWS IoT console**.

Python

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 pubsub.py --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --
cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-
endpoint
```

JavaScript

```
cd ~/aws-iot-device-sdk-js-v2/samples/node/pub_sub
node dist/index.js --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --
cert ~/certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-
endpoint
```

Troubleshoot problems with the sample application

If you encounter an error when you try to run the sample app, here are some things to check.

Check the certificate

If the certificate is not active, AWS IoT will not accept any connection attempts that use it for authorization. When creating your certificate, it's easy to overlook the **Activate** button. Fortunately, you can activate your certificate from the AWS IoT console.

To check your certificate's activation

1. In the AWS IoT console, in the left menu, choose **Secure**, and then choose **Certificates**.
2. In the list of certificates, find the certificate you created for the exercise and check its status in the **Status** column.

If you don't remember the certificate's name, check for any that are **Inactive** to see if they might be the one you're using.

Choose the certificate in the list to open its detail page. In the detail page, you can see its **Create date** to help you identify the certificate.

3. **To activate an inactive certificate**, from the certificate's detail page, choose **Actions** and then choose **Activate**.

If you found the correct certificate and it's active, but you're still having problems running the sample app, check its policy as the next step describes.

You can also try to create a new thing and a new certificate by following the steps in the section called "Create a thing object". If you create a new thing, you will need to give it a new thing name and download the new certificate files to your device.

Check the policy attached to the certificate

Policies authorize actions in AWS IoT. If the certificate used to connect to AWS IoT does not have a policy, or does not have a policy that allows it to connect, the connection will be refused, even if the certificate is active.

To check the policies attached to a certificate

1. Find the certificate as described in the previous item and open its details page.
2. In the left menu of the certificate's details page, choose **Policies** to see the policies attached to the certificate.
3. If there are no policies attached to the certificate, add one by choosing the **Actions** menu, and then choosing **Attach policy**.

   Choose the policy that you created earlier in the section called "Create AWS IoT resources".
4. If there is a policy attached, choose the policy tile to open its details page.

   In the details page, review the **Policy document** to make sure it contains the same information as the one you created in the section called "Create an AWS IoT policy".

Check the command line

Make sure you used the correct command line for your system. The commands used on Linux and macOS systems are often different from those used on Windows systems.

Check the endpoint address

Review the command you entered and double-check the endpoint address in your command to the one in your AWS IoT console.

Check the file names of the certificate files

Compare the file names in the command you entered to the file names of the certificate files in the `certs` directory.

Some systems might require the file names to be in quotes to work correctly.

Check the SDK installation

Make sure that your SDK installation is complete and correct.

If in doubt, reinstall the SDK on your device. In most cases, that's a matter of finding the section of the tutorial titled **Install the AWS IoT Device SDK for *`SDK language`*** and following the procedure again.

If you are using the **AWS IoT Device SDK for JavaScript**, remember to install the sample apps before you try to run them. Installing the SDK doesn't automatically install the sample apps. The sample apps must be installed manually after the SDK has been installed.

View MQTT messages with the AWS IoT MQTT client

This section describes how to use the AWS IoT MQTT test client in the AWS IoT console to watch the MQTT messages sent and received by AWS IoT. The example used in this section relates to the examples used in Getting started with AWS IoT Core tutorials; however, you can replace the *`topicName`* used in the examples with any topic name or topic filter used by your IoT solution.

Devices publish MQTT messages that are identified by topics to communicate their state to AWS IoT, and AWS IoT publishes MQTT messages to inform the devices and apps of changes and events. You can use the MQTT client to subscribe to these topics and watch the messages as they occur.

You can also use the MQTT test client to publish MQTT messages to subscribed devices and services in your AWS account.

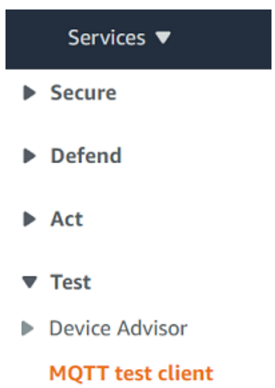
Contents

Viewing MQTT messages in the MQTT client

The following procedure explains how to subscribe to a specific MQTT topic that your device publishes messages to and view those messages in the AWS IoT console.

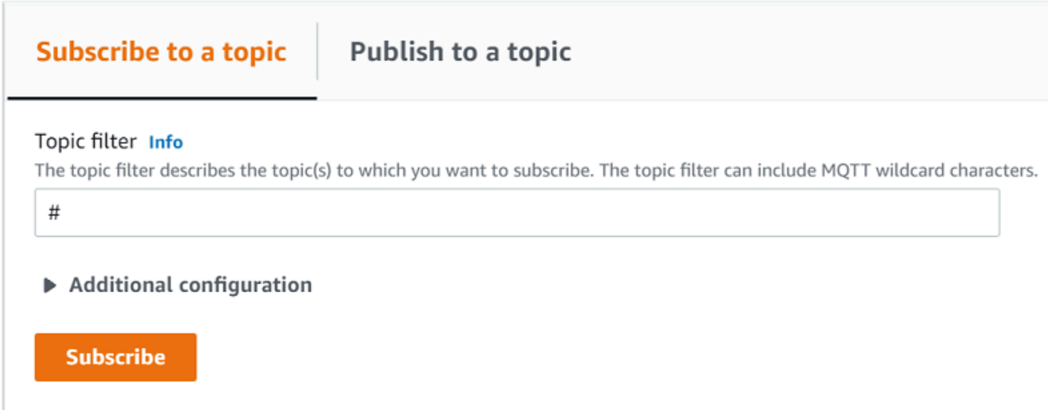
To view MQTT messages in the MQTT test client

1. In the AWS IoT console, in the left menu, choose **Test** and then choose **MQTT test client**.

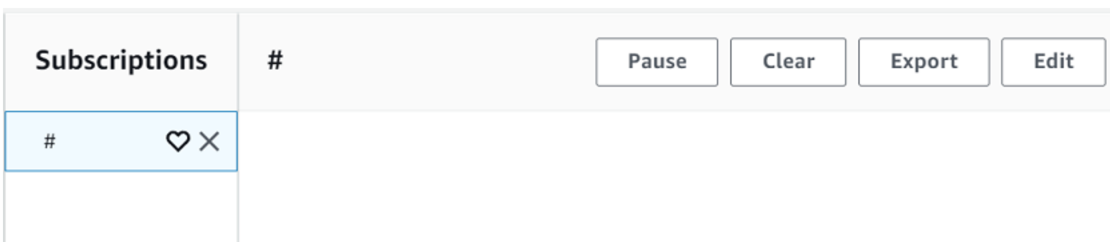


2. In the **Subscribe to a topic** tab, enter the *topicName* to subscribe to the topic on which your device publishes. For the getting started sample app, subscribe to **#**, which subscribes to all message topics.

   Continuing with the getting started example, on the **Subscribe to a topic** tab, in the **Topic filter** field, enter **#**, and then choose **Subscribe**.



The topic message log page, **#** opens and **#** appears in the **Subscriptions** list. If the device that you configured in [the section called "Configure your device"]() is running the example program, you should see the messages it sends to AWS IoT in the **#** message log. The message log entries will appear below the **Publish** section when messages with the subscribed topic are received by AWS IoT.



3. On the **#** message log page, you can also publish messages to a topic, but you'll need to specify the topic name. You cannot publish to the **#** topic.

   Messages published to subscribed topics appear in the message log as they are received, with the most recent message first.

Troubleshooting MQTT messages

Use the wild card topic filter

If your messages are not showing up in the message log as you expect, try subscribing to a wild card topic filter as described in [Topic name filters](). The MQTT multi-level wild card topic filter is the hash or pound sign (#) and can be used as the topic filter in the **Subscription topic** field.

Subscribing to the # topic filter subscribes to every topic received by the message broker. You can narrow the filter down by replacing elements of the topic filter path with a # multi-level wild card character or the '+' single-level wild-card character.

When using wild cards in a topic filter

- The multi-level wild card character must be the last character in the topic filter.
- The topic filter path can have only one single-level wild card character per topic level.

For example:

Topic filter	Displays messages with
`#`	Any topic name
`topic_1/#`	A topic name that starts with `topic_1/`
`topic_1/level_2/#`	A topic name that starts with `topic_1/level_2/`
`topic_1/+/level_3`	A topic name that starts with `topic_1/`, ends with `/level_3`, and has one element of any value in between.

For more information on topic filters, see Topic name filters.

Check for topic name errors

MQTT topic names and topic filters are case sensitive. If, for example, your device is publishing messages to `Topic_1` (with a capital *T*) instead of `topic_1`, the topic to which you subscribed, its messages would not appear in the MQTT test client. Subscribing to the wild card topic filter, however would show that the device is publishing messages and you could see that it was using a topic name that was not the one you expected.

Publishing MQTT messages from the MQTT client

To publish a message to an MQTT topic

1. On the MQTT test client page, in the **Publish to a topic** tab, in the **Topic name** field, enter the *`topicName`* of your message. In this example, use **`my/topic`**.

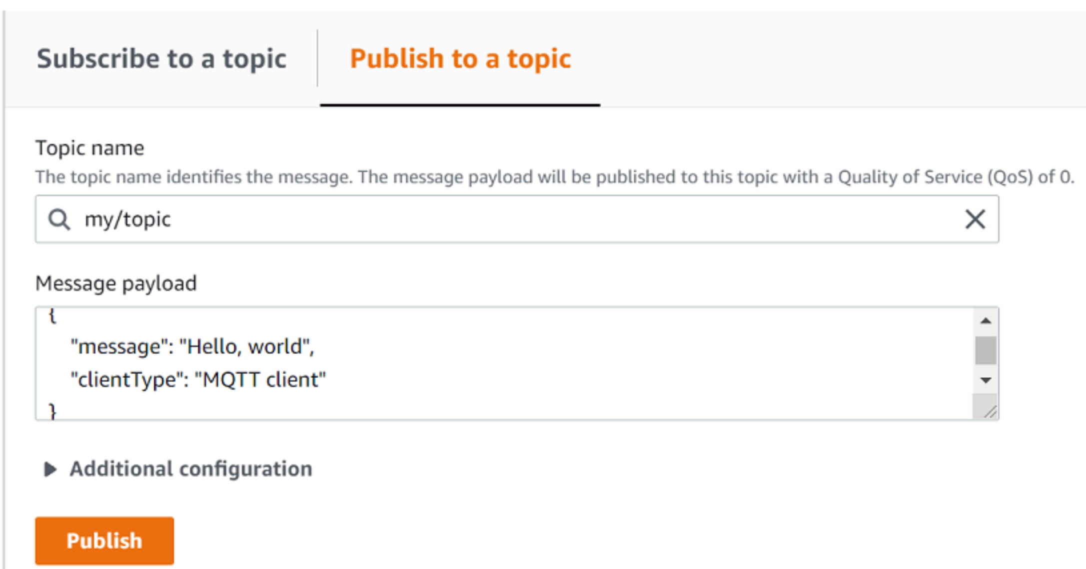
> **Note**
>
> Do not use personally identifiable information in topic names, whether using them in the MQTT test client or in your system implementation. Topic names can appear in unencrypted communications and reports.

2. In the message payload window, enter the following JSON:

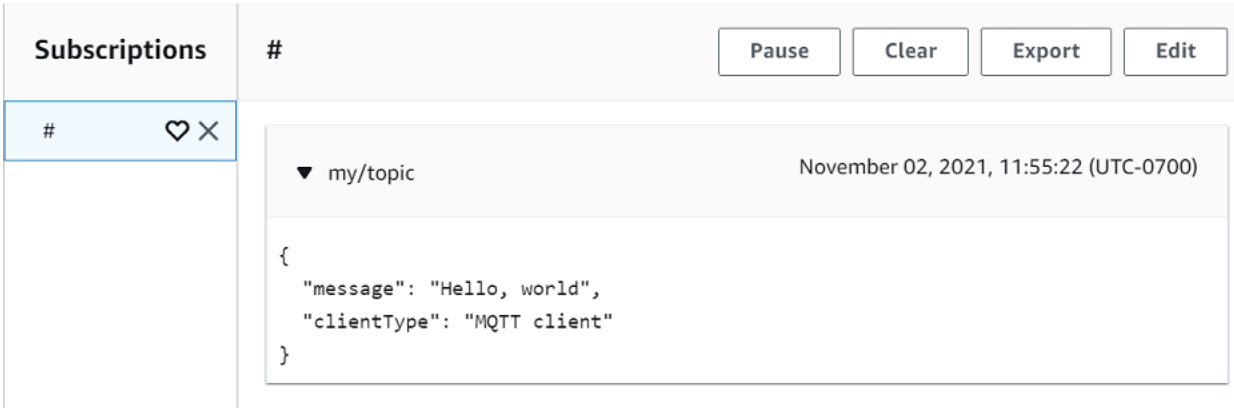
```
{
    "message": "Hello, world",
    "clientType": "MQTT test client"
}
```

3. Choose **Publish** to publish your message to AWS IoT.

> **Note**
>
> Make sure you are subscribed to the **my/topic** topic before publishing your message.



4. In the **Subscriptions** list, choose **my/topic** to see the message. You should see the message appear in the MQTT test client below the publish message payload window.



You can publish MQTT messages to other topics by changing the *topicName* in the **Topic name** field and choosing the **Publish** button.

> **Important**
>
> When you create multiple subscriptions with overlapping topics (e.g., probe1/temperature and probe1/#), there is a possibility that a single message published to a topic matching both subscriptions will be delivered multiple times, once for each overlapping subscription.

Testing Shared Subscriptions in the MQTT client

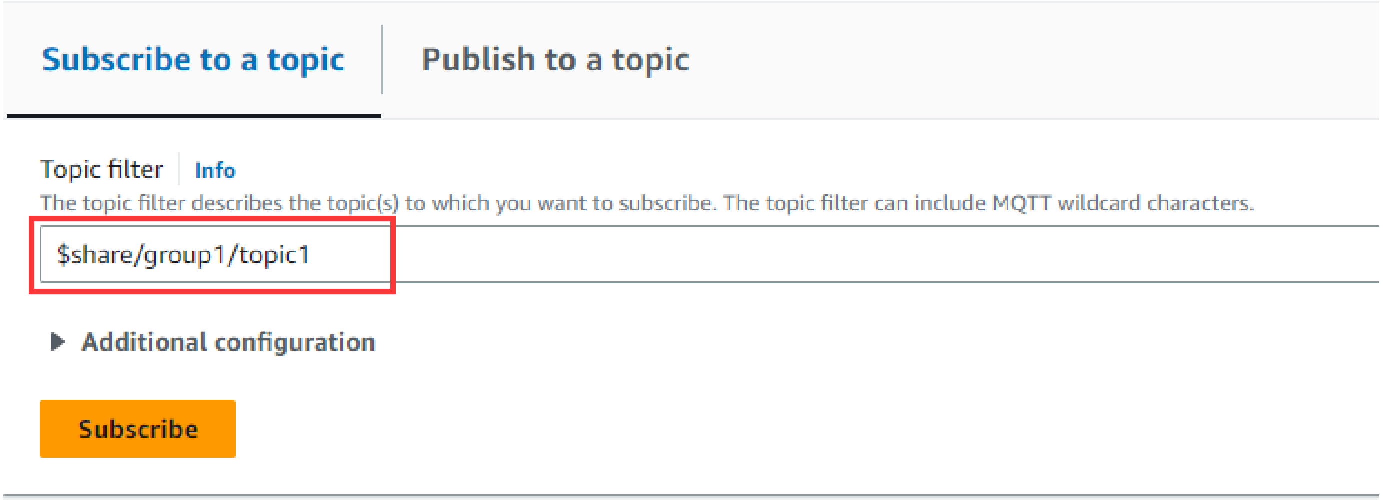
This section describes how to use the AWS IoT MQTT client in the AWS IoT console to watch the MQTT messages sent and received by AWS IoT using Shared Subscriptions. ??? allow multiple clients to share a subscription to a topic with only one client receiving messages published to that topic using a random distribution. To simulate multiple MQTT clients (in this example, two MQTT clients) sharing the same subscription, you open the AWS IoT MQTT client in the AWS IoT console from multiple web browsers. The example used in this section doesn't relate to the examples used in Getting started with AWS IoT Core tutorials. For more information, see Shared Subscriptions.

To share a subscription to an MQTT topic

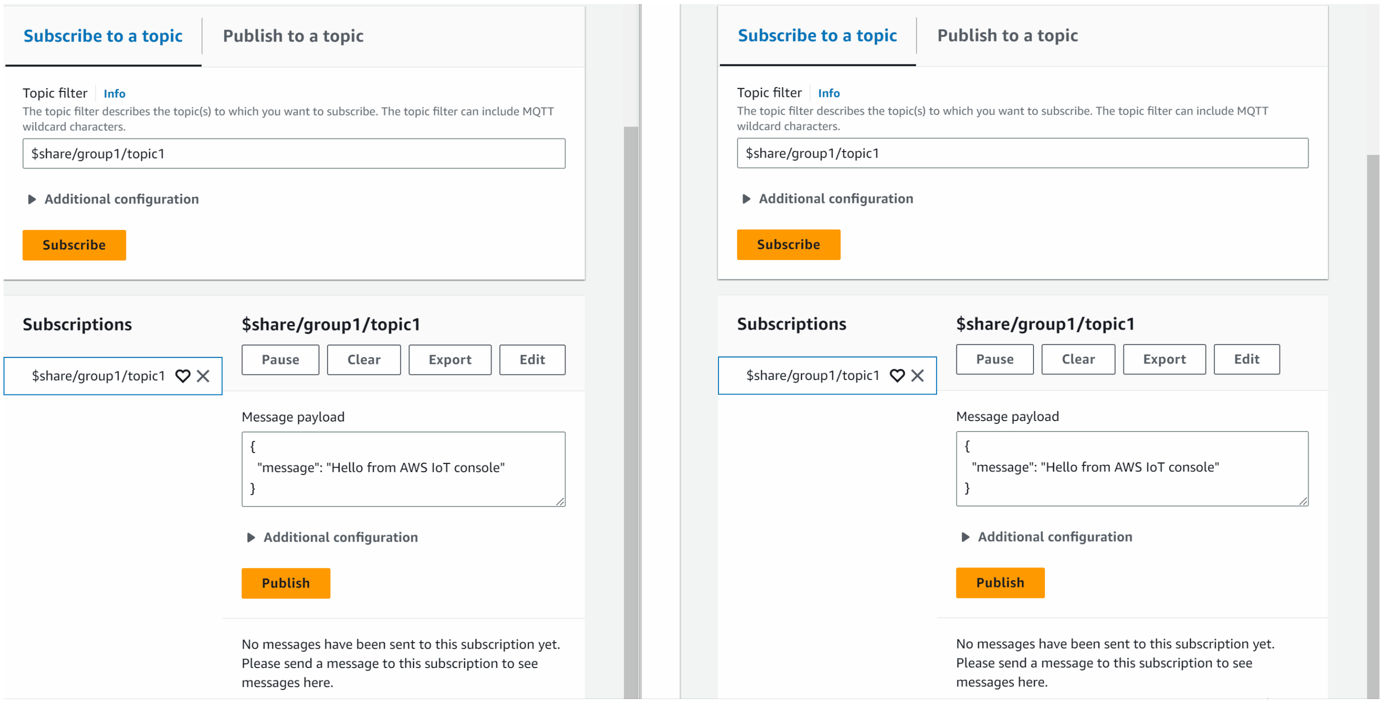
1. In the AWS IoT console, in the navigation pane, choose **Test** and then choose **MQTT test client**.
2. In the **Subscribe to a topic** tab, enter the *topicName* to subscribe to the topic on which your device publishes. To use Shared Subscriptions, subscribe to a Shared Subscription's topic filter as follows:

```
$share/{ShareName}/{TopicFilter}
```

An example topic filter can be **`$share/group1/topic1`**, which subscribes to the message topic **`topic1`**.



3. Open another web browser and repeat step1 and step2. In this way, you are simulating two different MQTT clients that share the same subscription **`$share/group1/topic1`**.
4. Choose one MQTT client, in the **Publish to a topic** tab, in the **Topic name** field, enter the *`topicName`* of your message. In this example, use **`topic1`**. Try publishing the message a few times. From the **Subscriptions** list of both MQTT clients, you should be able to see that the clients receive the message using a random distribution. In this example, we publish the same message "Hello from AWS IoT console" three times. The MQTT client on the left received the message twice and the MQTT client on the right received the message once.


Subscribe to a topic
Publish to a topic
Topic filter Info
The topic filter describes the topic(s) to which you want to subscribe. The topic filter can include MQTT wildcard characters.
$share/group1/topic1
Additional configuration
Subscribe
Subscriptions
$share/group1/topic1
Pause
Clear
Export
Edit
$share/group1/topic1
Message payload
{ "message": "Hello from AWS IoT console" }
Additional configuration
Publish
No messages have been sent to this subscription yet. Please send a message to this subscription to see messages here.

AWS IoT tutorials

The AWS IoT tutorials are divided into two learning paths to support two different goals. Choose the best learning path for your goal.

- **You want to build a proof-of-concept to test or demonstrate an AWS IoT solution idea**

  To demonstrate common IoT tasks and applications using the AWS IoT Device Client on your devices, follow the the section called “Building demos with the AWS IoT Device Client” learning path. The AWS IoT Device Client provides device software with which you can apply your own cloud resources to demonstrate an end-to-end solution with minimum development.

  For information about the AWS IoT Device Client, see the AWS IoT Device Client.

- **You want to learn how to build production software to deploy your solution**

  To create your own solution software that meets your specific requirements using an AWS IoT Device SDK, follow the the section called “Building solutions with the AWS IoT Device SDKs” learning path.

  For information about the available AWS IoT Device SDKs, see ???. For information about the AWS SDKs, see Tools to Build on AWS.

AWS IoT tutorial learning path options

- Building demos with the AWS IoT Device Client
- Building solutions with the AWS IoT Device SDKs

Building demos with the AWS IoT Device Client

The tutorials in this learning path walk you through the steps to develop demonstration software by using the AWS IoT Device Client. The AWS IoT Device Client provides software that runs on your IoT device to test and demonstrate aspects of an IoT solution that's built on AWS IoT.

The goal of these tutorials is to facilitate exploration and experimentation so you can feel confident that AWS IoT supports your solution before you develop your device software.

What you'll learn in these tutorials:

- How to prepare a Raspberry Pi for use as an IoT device with AWS IoT
- How to demonstrate AWS IoT features by using the AWS IoT Device Client on your device

In this learning path, you'll install the AWS IoT Device Client on your own Raspberry Pi and create the AWS IoT resources in the cloud to demonstrate IoT solution ideas. While the tutorials in this learning path demonstrate features by using a Raspberry Pi, they explain the goals and procedures to help you adapt them to other devices.

Prerequisites to building demos with the AWS IoT Device Client

This section describes what you'll need to have before you start the tutorials in this learning path.

To complete the tutorials in this learning path, you'll need:

- **An AWS account**

  You can use your existing AWS account, if you have one, but you might need to add additional roles or permissions to use the AWS IoT features these tutorials use.

  If you need to create a new AWS account, see the section called “Set up AWS account”.
- **A Raspberry Pi or compatible IoT device**

  The tutorials use a Raspberry Pi because it comes in different form factors, it's ubiquitous, and it's a relatively inexpensive demonstration device. The tutorials have been tested on the Raspberry Pi 3 Model B+, the Raspberry Pi 4 Model B, and on an Amazon EC2 instance running Ubuntu Server 20.04 LTS (HVM). To use the AWS CLI and run the commands, We recommend that you use the latest version of the Raspberry Pi OS (Raspberry Pi OS (64-bit) or the OS Lite). Earlier versions of the OS might work, but we haven't tested it.

  > **Note**
  >
  > The tutorials explain the goals of each step to help you adapt them to IoT hardware that we haven't tried them on; however, they do not specifically describe how to adapt them to other devices.
- **Familiarity with the IoT device's operating system**

The steps in these tutorials assume that you are familiar with using basic Linux commands and operations from the command line interface supported by a Raspberry Pi. If you're not familiar with these operations, you might want to give yourself more time to complete the tutorials.

To complete these tutorials, you should already understand how to:

- Safely perform basic device operations such as assembling and connecting components, connecting the device to required power sources, and installing and removing memory cards.
- Upload and download system software and files to the device. If your device doesn't use a removable storage device, such as a microSD card, you'll need to know how to connect to your device and upload and download system software and files to the device.
- Connect your device to the networks that you plan to use it on.
- Connect to your device from another computer using an SSH terminal or similar program.
- Use a command line interface to create, copy, move, rename, and set the permissions of files and directories on the device.
- Install new programs on the device.
- Transfer files to and from your device using tools such as FTP or SCP.

- **A development and testing environment for your IoT solution**

  The tutorials describe the software and hardware required; however, the tutorials assume that you'll be able to perform operations that might not be described explicitly. Examples of such hardware and operations include:

  - **A local host computer to download and store files on**

    For the Raspberry Pi, this is usually a personal computer or laptop that can read and write to microSD memory cards. The local host computer must:

    - Be connected to the Internet.
    - Have the AWS CLI installed and configured.
    - Have a web browser that supports the AWS console.
  - **A way to connect your local host computer to your device to communicate with it, to enter commands, and to transfer files**

    On the Raspberry Pi, this is often done using SSH and SCP from the local host computer.
  - **A monitor and keyboard to connect to your IoT device**

    These can be helpful, but are not required to complete the tutorials.

- **A way for your local host computer and your IoT devices to connect to the internet**

  This could be a cabled or a wireless network connection to a router or gateway that's connected to the internet. The local host must also be able to connect to the Raspberry Pi. This might require them to be on the same local area network. The tutorials can't show you how to set this up for your particular device or device configuration, but they show how you can test this connectivity.

- **Access to your local area network's router to view the connected devices**

  To complete the tutorials in this learning path, you'll need to be able to find the IP address of your IoT device.

  On a local area network, this can be done by accessing the admin interface of the network router your devices connect to. If you can assign a fixed IP address for your device in the router, you can simplify reconnection after each time the device restarts.

  If you have a keyboard and a monitor attached to the device, **ifconfig** can display the device's IP address.

  If none of these are an option, you'll need to find a way to identify the device's IP address after each time it restarts.

After you have all your materials, continue to the section called "Preparing to use IoT Device Client".

Tutorials in this learning path

- Tutorial: Preparing your devices for the AWS IoT Device Client
- Tutorial: Installing and configuring the AWS IoT Device Client
- Tutorial: Demonstrate MQTT message communication with the AWS IoT Device Client
- Tutorial: Demonstrate remote actions (jobs) with the AWS IoT Device Client
- Tutorial: Cleaning up after running the AWS IoT Device Client tutorials

Tutorial: Preparing your devices for the AWS IoT Device Client

This tutorial walks you through the initialization of your Raspberry Pi to prepare it for the subsequent tutorials in this learning path.

The goal of this tutorial is to install the current version of the device's operating system and make sure that you can communicate with your device in the context of your development environment.

Prerequisites

Before you start this tutorial, make sure that you have the items listed in the section called "Prerequisites to building demos with the AWS IoT Device Client" available and ready to use.

This tutorial takes about 90 minutes to complete.

In this tutorial, you'll:

- Install and update the operating system of your device.
- Install and verify any additional software needed to run the tutorials.
- Test your device's connectivity and install the required certificates.

After you complete this tutorial, the next tutorial prepares your device for the demos that use the AWS IoT Device Client.

Procedures in this tutorial

- Install and update the operating system of the device
- Install and verify required software on your device
- Test your device and save the Amazon CA cert

Install and update the operating system of the device

The procedures in this section describe how to initialize the microSD card that the Raspberry Pi uses for its system drive. The Raspberry Pi's microSD card contains its operating system (OS) software as well as space for its application file storage. If you're not using a Raspberry Pi, follow the device's instructions to install and update the device's operating system software.

After you complete this section, you should be able to start your IoT device and connect to it from the terminal program on your local host computer.

Required equipment:

- Your local development and testing environment
- A Raspberry Pi that or your IoT device, that can connect to the internet

- A microSD memory card with at least 8 GB capacity or sufficient storage for the OS and required software.

> **Note**
>
> When selecting a microSD card for these exercises, choose one that is as large as necessary but, as small as possible.
> A small SD card will be faster to back up and update. On the Raspberry Pi, you won't need more than an 8-GB microSD card for these tutorials. If you need more space for your specific application, the smaller image files you save in these tutorials can resize the file system on a larger card to use all the supported space of the card you choose.

Optional equipment:

- A USB keyboard connected to the Raspberry Pi
- An HDMI monitor and cable to connect the monitor to the Raspberry Pi

Procedures in this section:

- Load the device's operating system onto microSD card
- Start your IoT device with the new operating system
- Connect your local host computer to your device

Load the device's operating system onto microSD card

This procedure uses the local host computer to load the device's operating system onto a microSD card.

> **Note**
>
> If your device doesn't use a removable storage medium for its operating system, install the operating system using the procedure for that device and continue to the section called "Start your IoT device".

To install the operating system on your Raspberry Pi

1. On your local host computer, download and unzip the Raspberry Pi operating system image that you want to use. The latest versions are available from https://www.raspberrypi.com/software/operating-systems/

   Choosing a version of Raspberry Pi OS

   This tutorial uses the **Raspberry Pi OS Lite** version because it's the smallest version that supports these the tutorials in this learning path. This version of the Raspberry Pi OS has only a command line interface and doesn't have a graphical user interface. A version of the latest Raspberry Pi OS with a graphical user interface will also work with these tutorials; however, the procedures described in this learning path use only the command line interface to perform operations on the Raspberry Pi.
2. Insert your microSD card into the local host computer.
3. Using an SD card imaging tool, write the unzipped OS image file to the microSD card.
4. After writing the Raspberry Pi OS image to the microSD card:

   a. Open the BOOT partition on the microSD card in a command line window or file explorer window.

   b. In the BOOT partition of the microSD card, in the root directory, create an empty file named `ssh` with no file extension and no content. This tells the Raspberry Pi to enable SSH communications the first time it starts.
5. Eject the microSD card and safely remove it from the local host computer.

Your microSD card is ready to the section called "Start your IoT device".

Start your IoT device with the new operating system

This procedure installs the microSD card and starts your Raspberry Pi for the first time using the downloaded operating system.

To start your IoT device with the new operation system

1. With the power disconnected from the device, insert the microSD card from the previous step, the section called "Load the OS", into the Raspberry Pi.
2. Connect the device to a wired network.

3. These tutorials will interact with your Raspberry Pi from your local host computer using an SSH terminal.

   If you also want to interact with the device directly, you can:

   a. Connect an HDMI monitor to it to watch the Raspberry Pi's console messages before you can connect the terminal window on your local host computer to your Raspberry Pi.

   b. Connect a USB keyboard to it if you want to interact directly with the Raspberry Pi.

4. Connect the power to the Raspberry Pi and wait about a minute for it to initialize.

   If you have a monitor connected to your Raspberry Pi, you can watch the start-up process on it.

5. Find out your device's IP address:

   - If you connected an HDMI monitor to the Raspberry Pi, the IP address appears in the messages displayed on the monitor
   - If you have access to the router your Raspberry Pi is connects to, you can see its address in the router's admin interface.

After you have your Raspberry Pi's IP address, you're ready to the section called "Connect your host computer".

Connect your local host computer to your device

This procedure uses the terminal program on your local host computer to connect to your Raspberry Pi and change its default password.

To connect your local host computer to your device

1. On your local host computer, open the SSH terminal program:

   - Windows: PuTTY
   - Linux/macOS: `Terminal`

   > **Note**
   >
   > PuTTY isn't installed automatically on Windows. If it's not on your computer, you might need to download and install it.

2. Connect the terminal program to your Raspberry Pi’s IP address and log in using its default credentials.

```
username: pi
password: raspberry
```

3. After you log in to your Raspberry Pi, change the password for the `pi` user.

```
passwd
```

Follow the prompts to change the password.

```
Changing password for pi.
Current password: raspberry
New password: YourNewPassword
Retype new password: YourNewPassword
passwd: password updated successfully
```

After you have the Raspberry Pi's command line prompt in the terminal window and changed the password, you're ready to continue to the section called “Install and verify required software”.

Install and verify required software on your device

The procedures in this section continue from the previous section to bring your Raspberry Pi's operating system up to date and install the software on the Raspberry Pi that will be used in the next section to build and install the AWS IoT Device Client.

After you complete this section, your Raspberry Pi will have an up-to-date operating system, the software required by the tutorials in this learning path, and it will be configured for your location.

Required equipment:

- Your local development and testing environment from the previous section
- The Raspberry Pi that you used in the previous section
- The microSD memory card from the previous section

> **Note**
>
> The Raspberry Pi Model 3+ and Raspberry Pi Model 4 can perform all the commands described in this learning path. If your IoT device can't compile software or run the AWS Command Line Interface, you might need to install the required compilers on your local host computer to build the software and then transfer it to your IoT device. For more information about how to install and build software for your device, see the documentation for your device's software.

Procedures in this section:

- Update operating system software
- Install the required applications and libraries
- (Optional) Save the microSD card image

Update operating system software

This procedure updates the operating system software.

To update the operating system software on the Raspberry Pi

Perform these steps in the terminal window of your local host computer.

1. Enter these commands to update the system software on your Raspberry Pi.

   ```
   sudo apt-get -y update
   sudo apt-get -y upgrade
   sudo apt-get -y autoremove
   ```

2. Update the Raspberry Pi's locale and time zone settings (optional).

   Enter this command to update the device's locale and time zone settings.

   ```
   sudo raspi-config
   ```

   a. To set the device's locale:

      i. In the **Raspberry Pi Software Configuration Tool (raspi-config)** screen, choose option **5**.

```
5 Localisation Options Configure language and regional settings
```

Use the **Tab** key to move to **<Select>,** and then press the **space bar**.

ii. In the localization options menu, choose option **L1**.

```
L1 Locale Configure language and regional settings
```

Use the **Tab** key to move to **<Select>,** and then press the **space bar**.

iii. In the list of locale options, choose the locales that you want to install on your Raspberry Pi by using the arrow keys to scroll and the **space bar** to mark those that you want.

In the United States, **`en_US.UTF-8`** is a good one to choose.

iv. After selecting the locales for your device, use the **Tab** key to choose **<OK>**, and then press the **space bar** to display the **Configuring locales** confirmation page.

b. To set the device's time zone:

i. In the **raspi-config** screen, choose option **5**.

```
5 Localisation Options Configure language and regional settings
```

Use the **Tab** key to move to **<Select>,** and then press the **space bar**.

ii. In the localization options menu, use the arrow key to choose option **L2**:

```
L2 time zone Configure time zone
```

Use the **Tab** key to move to **<Select>,** and then press the **space bar**.

iii. In the **Configuring tzdata** menu, choose your geographical area from the list.

Use the **Tab** key to move to **<OK>**, and then press the **space bar**.

iv. In the list of cities, use the arrow keys to choose a city in your time zone.

To set the time zone, use the **Tab** key to move to **<OK>**, and then press the **space bar**.

c. When you've finished updating the settings, use the **Tab** key to move to **<Finish>**, and then press the **space bar** to close the **raspi-config** app.

3. Enter this command to restart your Raspberry Pi.

```
sudo shutdown -r 0
```

4. Wait for your Raspberry Pi to restart.
5. After your Raspberry Pi has restarted, reconnect the terminal window on your local host computer to your Raspberry Pi.

Your Raspberry Pi system software is now configured and you're ready to continue to the section called "Install applications and libraries".

Install the required applications and libraries

This procedure installs the application software and libraries that the subsequent tutorials use.

If you are using a Raspberry Pi, or if you can compile the required software on your IoT device, perform these steps in the terminal window on your local host computer. If you must compile software for your IoT device on your local host computer, review the software documentation for your IoT device for information about how to do these steps on your device.

To install the application software and libraries on your Raspberry Pi

1. Enter this command to install the application software and libraries.

```
sudo apt-get -y install build-essential libssl-dev cmake unzip git python3-pip
```

2. Enter these commands to confirm that the correct version of the software was installed.

```
gcc --version
cmake --version
openssl version
git --version
```

3. Confirm that these versions of the application software are installed:

   - `gcc`: 9.3.0 or later
   - `cmake`: 3.10.x or later
   - `OpenSSL`: 1.1.1 or later
   - `git`: 2.20.1 or later

If your Raspberry Pi has acceptable versions of the required application software, you're ready to continue to the section called "(Optional) Save microSD image".

(Optional) Save the microSD card image

Throughout the tutorials in this learning path, you'll encounter these procedures to save a copy of the Raspberry Pi's microSD card image to a file on your local host computer. While encouraged, they are not required tasks. By saving the microSD card image where suggested, you can skip the procedures that precede the save point in this learning path, which can save time if you find the need to retry something. The consequence of not saving the microSD card image periodically is that you might have to restart the tutorials in the learning path from the beginning if your microSD card is damaged or if you accidentally configure an app or its settings incorrectly.

At this point, your Raspberry Pi's microSD card has an updated OS and the basic application software loaded. You can save the time it took you to complete the preceding steps by saving the contents of the microSD card to a file now. Having the current image of your device's microSD card image lets you start from this point to continue or retry a tutorial or procedure without the need to install and update the software from scratch.

To save the microSD card image to a file

1. Enter this command to shut down the Raspberry Pi.

   ```
   sudo shutdown -h 0
   ```

2. After the Raspberry Pi shuts down completely, remove its power.
3. Remove the microSD card from the Raspberry Pi.
4. On your local host computer:

   a. Insert the microSD card.

   b. Using your SD card imaging tool, save the microSD card's image to a file.

   c. After the microSD card's image has been saved, eject the card from the local host computer.

5. With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.
6. Apply power to the Raspberry Pi.

7. After waiting about a minute, on the local host computer, reconnect the terminal window on your local host computer that was connected to your Raspberry Pi., and then log in to the Raspberry Pi.

Test your device and save the Amazon CA cert

The procedures in this section continue from the previous section to install the AWS Command Line Interface and the Certificate Authority certificate used to authenticate your connections with AWS IoT Core.

After you complete this section, you'll know that your Raspberry Pi has the necessary system software to install the AWS IoT Device Client and that it has a working connection to the internet.

Required equipment:

- Your local development and testing environment from the previous section
- The Raspberry Pi that you used in the previous section
- The microSD memory card from the previous section

Procedures in this section:

- Install the AWS Command Line Interface
- Configure your AWS account credentials
- Download the Amazon Root CA certificate
- (Optional) Save the microSD card image

Install the AWS Command Line Interface

This procedure installs the AWS CLI onto your Raspberry Pi.

If you are using a Raspberry Pi or if you can compile software on your IoT device, perform these steps in the terminal window on your local host computer. If you must compile software for your IoT device on your local host computer, review the software documentation for your IoT device for information about the libraries it requires.

To install the AWS CLI on your Raspberry Pi

1. Run these commands to download and install the AWS CLI.

```
export PATH=$PATH:~/.local/bin # configures the path to include the directory with
 the AWS CLI
git clone https://github.com/aws/aws-cli.git # download the AWS CLI code from
 GitHub
cd aws-cli && git checkout v2 # go to the directory with the repo and checkout
 version 2
pip3 install -r requirements.txt # install the prerequisite software
```

2. Run this command to install the AWS CLI. This command can take up to 15 minutes to complete.

```
pip3 install . # install the AWS CLI
```

3. Run this command to confirm that the correct version of the AWS CLI was installed.

```
aws --version
```

   The version of the AWS CLI should be 2.2 or later.

If the AWS CLI displayed its current version, you're ready to continue to the section called "Configure account credentials".

Configure your AWS account credentials

In this procedure, you'll obtain AWS account credentials and add them for use on your Raspberry Pi.

To add your AWS account credentials to your device

1. Obtain an **Access Key ID** and **Secret Access Key** from your AWS account to authenticate the AWS CLI on your device.

   If you're new to AWS IAM, https://aws.amazon.com/premiumsupport/knowledge-center/create-access-key/ describes the process to run in the AWS console to create AWS IAM credentials to use on your device.

2. In the terminal window on your local host computer that's connected to your Raspberry Pi. and with the **Access Key ID** and **Secret Access Key** credentials for your device:

   a. Run the AWS configure app with this command:

```
aws configure
```

b. Enter your credentials and configuration information when prompted:

```
AWS Access Key ID: your Access Key ID
AWS Secret Access Key: your Secret Access Key
Default region name: your AWS Region code
Default output format: json
```

3. Run this command to test your device's access to your AWS account and AWS IoT Core endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

It should return your AWS account-specific AWS IoT data endpoint, such as this example:

```
{
    "endpointAddress": "a3EXAMPLEffp-ats.iot.us-west-2.amazonaws.com"
}
```

If you see your AWS account-specific AWS IoT data endpoint, your Raspberry Pi has the connectivity and permissions to continue to the section called “Download the Amazon Root CA certificate”.

Important

Your AWS account credentials are now stored on the microSD card in your Raspberry Pi. While this makes future interactions with AWS easy for you and the software you'll create in these tutorials, they will also be saved and duplicated in any microSD card images you make after this step by default.

To protect the security of your AWS account credentials, before you save any more microSD card images, consider erasing the credentials by running `aws configure` again and entering random characters for the **Access Key ID** and **Secret Access Key** to prevent your AWS account credentials from compromised.

If you find that you have saved your AWS account credentials inadvertently, you can deactivate them in the AWS IAM console.

Download the Amazon Root CA certificate

This procedure downloads and saves a copy of a certificate of the Amazon Root Certificate Authority (CA). Downloading this certificate saves it for use in the subsequent tutorials and it also tests your device's connectivity with AWS services.

To download and save the Amazon Root CA certificate

1. Run this command to create a directory for the certificate.

   ```
   mkdir ~/certs
   ```

2. Run this command to download the Amazon Root CA certificate.

   ```
   curl -o ~/certs/AmazonRootCA1.pem https://www.amazontrust.com/repository/
   AmazonRootCA1.pem
   ```

3. Run these commands to set the access to the certificate directory and its file.

   ```
   chmod 745 ~
   chmod 700 ~/certs
   chmod 644 ~/certs/AmazonRootCA1.pem
   ```

4. Run this command to see the CA certificate file in the new directory.

   ```
   ls -l ~/certs
   ```

   You should see an entry like this. The date and time will be different; however, the file size and all other info should be the same as shown here.

   ```
   -rw-r--r-- 1 pi pi 1188 Oct 28 13:02 AmazonRootCA1.pem
   ```

   If the file size is not 1188, check the **curl** command parameters. You might have downloaded an incorrect file.

(Optional) Save the microSD card image

At this point, your Raspberry Pi's microSD card has an updated OS and the basic application software loaded.

To save the microSD card image to a file

1. In the terminal window on your local host computer, clear your AWS credentials.

    a. Run the AWS configure app with this command:

    ```
    aws configure
    ```

    b. Replace your credentials when prompted. You can leave **Default region name** and **Default output format** as they are by pressing **Enter**.

    ```
    AWS Access Key ID [****************YT2H]: XYXYXYXYX
    AWS Secret Access Key [****************9plH]: XYXYXYXYX
    Default region name [us-west-2]:
    Default output format [json]:
    ```

2. Enter this command to shut down the Raspberry Pi.

    ```
    sudo shutdown -h 0
    ```

3. After the Raspberry Pi shuts down completely, remove its power connector.
4. Remove the microSD card from your device.
5. On your local host computer:

    a. Insert the microSD card.

    b. Using your SD card imaging tool, save the microSD card's image to a file.

    c. After the microSD card's image has been saved, eject the card from the local host computer.
6. With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.
7. Apply power to the device.
8. After about a minute, on the local host computer, restart the terminal window session and log in to the device.

    Don't reenter your AWS account credentials yet.

After you have restarted and logged in to your Raspberry Pi, you're ready to continue to the section called "Installing and configuring IoT device client".

Tutorial: Installing and configuring the AWS IoT Device Client

This tutorial walks you through the installation and configuration of the AWS IoT Device Client and the creation of AWS IoT resources that you'll use in this and other demos.

To start this tutorial:

- Have your local host computer and Raspberry Pi from the previous tutorial ready.

This tutorial can take up to 90 minutes to complete.

When you're finished with this topic:

- Your IoT device will be ready to use in other AWS IoT Device Client demos.
- You'll have provisioned your IoT device in AWS IoT Core.
- You'll have downloaded and installed the AWS IoT Device Client on your device.
- You'll have saved an image of your device's microSD card that can be used in subsequent tutorials.

Required equipment:

- Your local development and testing environment from the previous section
- The Raspberry Pi that you used in the previous section
- The microSD memory card from the Raspberry Pi that you used in the previous section

Procedures in this tutorial

- Download and save the AWS IoT Device Client
- Provision your Raspberry Pi in AWS IoT
- Configure the AWS IoT Device Client to test connectivity

Download and save the AWS IoT Device Client

The procedures in this section download the AWS IoT Device Client, compile it, and install it on your Raspberry Pi. After you test the installation, you can save the image of the Raspberry Pi's microSD card to use later when you want to try the tutorials again.

Procedures in this section:

- Download and build the AWS IoT Device Client
- Create the directories used by the tutorials
- (Optional) Save the microSD card image

Download and build the AWS IoT Device Client

This procedure installs the AWS IoT Device Client on your Raspberry Pi.

Perform these commands in the terminal window on your local host computer that is connected to your Raspberry Pi.

To install the AWS IoT Device Client on your Raspberry Pi

1. Enter these commands to download and build the AWS IoT Device Client on your Raspberry Pi.

```
cd ~
git clone https://github.com/awslabs/aws-iot-device-client aws-iot-device-client
mkdir ~/aws-iot-device-client/build && cd ~/aws-iot-device-client/build
cmake ../
```

2. Run this command to build the AWS IoT Device Client. This command can take up to 15 minutes to complete.

```
cmake --build . --target aws-iot-device-client
```

   The warning messages displayed as the AWS IoT Device Client compiles can be ignored.

   These tutorials have been tested with the AWS IoT Device Client built on **gcc**, version (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110 on the Oct 30th 2021 version of Raspberry Pi OS (bullseye) on **gcc**, version (Raspbian 8.3.0-6+rpi1) 8.3.0 on the May 7th 2021 version of the Raspberry Pi OS (buster).

3. After the AWS IoT Device Client finishes building, test it by running this command.

```
./aws-iot-device-client --help
```

If you see the command line help for the AWS IoT Device Client, the AWS IoT Device Client has been built successfully and is ready for you to use.

Create the directories used by the tutorials

This procedure creates the directories on the Raspberry Pi that will be used to store the files used by the tutorials in this learning path.

To create the directories used by the tutorials in this learning path:

1. Run these commands to create the required directories.

```
mkdir ~/dc-configs
mkdir ~/policies
mkdir ~/messages
mkdir ~/certs/testconn
mkdir ~/certs/pubsub
mkdir ~/certs/jobs
```

2. Run these commands to set the permissions on the new directories.

```
chmod 745 ~
chmod 700 ~/certs/testconn
chmod 700 ~/certs/pubsub
chmod 700 ~/certs/jobs
```

After you create these directories and set their permission, continue to the section called "(Optional) Save the microSD card image".

(Optional) Save the microSD card image

At this point, your Raspberry Pi's microSD card has an updated OS, the basic application software, and the AWS IoT Device Client.

If you want to come back to try these exercises and tutorials again, you can skip the preceding procedures by writing the microSD card image that you save with this procedure to a new microSD card and continue the tutorials from the section called "Provision your Raspberry Pi".

To save the microSD card image to a file:

In the terminal window on your local host computer that's connected to your Raspberry Pi:

1. Confirm that your AWS account credentials have not been stored.

   a. Run the AWS configure app with this command:

```
aws configure
```

b. If your credentials have been stored (if they are displayed in the prompt), then enter the **XYXYXYXYX** string when prompted as shown here. Leave **Default region name** and **Default output format** blank.

```
AWS Access Key ID [***************YXYX]: XYXYXYXYX
AWS Secret Access Key [***************YXYX]: XYXYXYXYX
Default region name:
Default output format:
```

2. Enter this command to shutdown the Raspberry Pi.

```
sudo shutdown -h 0
```

3. After the Raspberry Pi shuts down completely, remove its power connector.
4. Remove the microSD card from your device.
5. On your local host computer:

   a. Insert the microSD card.

   b. Using your SD card imaging tool, save the microSD card’s image to a file.

   c. After the microSD card’s image has been saved, eject the card from the local host computer.

You can continue with this microSD card in the section called “Provision your Raspberry Pi”.

Provision your Raspberry Pi in AWS IoT

The procedures in this section start with the saved microSD image that has the AWS CLI and AWS IoT Device Client installed and create the AWS IoT resources and device certificates that provision your Raspberry Pi in AWS IoT.

Install the microSD card in your Raspberry Pi

This procedure installs the microSD card with the necessary software loaded and configured into the Raspberry Pi and configures your AWS account so that you can continue with the tutorials in this learning path.

Use a microSD card from the section called “(Optional) Save the microSD card image” that has the necessary software for the exercises and tutorials in this learning path.

To install the microSD card in your Raspberry Pi

1. With the power disconnected from the Raspberry Pi, insert the microSD card into the Raspberry Pi.
2. Apply power to the Raspberry Pi.
3. After about a minute, on the local host computer, restart the terminal window session and log in to the Raspberry Pi.
4. On your local host computer, in the terminal window, and with the **Access Key ID** and **Secret Access Key** credentials for your Raspberry Pi:

   a. Run the AWS configure app with this command:

   ```
   aws configure
   ```

   b. Enter your AWS account credentials and configuration information when prompted:

   ```
   AWS Access Key ID [***************YXYX]: your Access Key ID
   AWS Secret Access Key [***************YXYX]: your Secret Access Key
   Default region name [us-west-2]: your AWS Region code
   Default output format [json]: json
   ```

After you have restored your AWS account credentials, you're ready to continue to the section called “Provision your device in AWS IoT Core”.

Provision your device in AWS IoT Core

The procedures in this section create the AWS IoT resources that provision your Raspberry Pi in AWS IoT. As you create these resources, you'll be asked to record various pieces of information. This information is used by the AWS IoT Device Client configuration in the next procedure.

For your Raspberry Pi to work with AWS IoT, it must be provisioned. Provisioning is the process of creating and configuring the AWS IoT resources that are necessary to support your Raspberry Pi as an IoT device.

With your Raspberry Pi powered up and restarted, connect the terminal window on your local host computer to the Raspberry Pi and complete these procedures.

Procedures in this section:

- Create and download device certificate files
- Create AWS IoT resources

Create and download device certificate files

This procedure creates the device certificate files for this demo.

To create and download the device certificate files for your Raspberry Pi

1. In the terminal window on your local host computer, enter these commands to create the device certificate files for your device.

```
mkdir ~/certs/testconn
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pem-outfile "~/certs/testconn/device.pem.crt" \
--public-key-outfile "~/certs/testconn/public.pem.key" \
--private-key-outfile "~/certs/testconn/private.pem.key"
```

   The command returns a response like the following. Record the *certificateArn* value for later use.

```
{
    "certificateArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fcd810f3aea2d92abc227d269",
    "certificateId":
 "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fcd810f3aea2d92abc227d269",
    "certificatePem": "-----BEGIN CERTIFICATE-----
\nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n-----END CERTIFICATE-----
\n",
    "keyPair": {
        "PublicKey": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n-----END PUBLIC KEY-----
\n",
        "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----
\nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n-----END RSA PRIVATE KEY-----\n"
    }
}
```

2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 745 ~
chmod 700 ~/certs/testconn
chmod 644 ~/certs/testconn/*
chmod 600 ~/certs/testconn/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

   ```
   ls -l ~/certs/testconn
   ```

   The output of the command should be the same as what you see here, except the file dates and times will be different.

   ```
   -rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt
   -rw------- 1 pi pi 1675 Oct 28 13:02 private.pem.key
   -rw-r--r-- 1 pi pi  451 Oct 28 13:02 public.pem.key
   ```

At this point, you have the device certificate files installed on your Raspberry Pi and you can continue to the section called "Create AWS IoT resources".

Create AWS IoT resources

This procedure provisions your device in AWS IoT by creating the resources that your device needs to access AWS IoT features and services.

To provision your device in AWS IoT

1. In the terminal window on your local host computer, enter the following command to get the address of the device data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type IoT:Data-ATS
   ```

   The command from the previous steps returns a response like the following. Record the *endpointAddress* value for later use.

   ```
   {
       "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. Enter this command to create an AWS IoT thing resource for your Raspberry Pi.

```
aws iot create-thing --thing-name "DevCliTestThing"
```

If your AWS IoT thing resource was created, the command returns a response like this.

```
{
    "thingName": "DevCliTestThing",
    "thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/DevCliTestThing",
    "thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"
}
```

3. In the terminal window:

   a. Open a text editor, such as nano.

   b. Copy this JSON policy document and paste it into your open text editor.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Subscribe",
                "iot:Receive",
                "iot:Connect"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

> **Note**
>
> This policy document generously grants every resource permission to connect, receive, publish, and subscribe. Normally policies grant only permission to specific resources to perform specific actions. However, for the initial device connectivity

test, this overly general and permissive policy is used to minimize the chance of an access problem during this test. In the subsequent tutorials, more narrowly scoped policy documents will be use to demonstrate better practices in policy design.

c. Save the file in your text editor as **~/policies/dev_cli_test_thing_policy.json**.

4. Run this command to use the policy document from the previous steps to create an AWS IoT policy.

```
aws iot create-policy \
--policy-name "DevCliTestThingPolicy" \
--policy-document "file://~/policies/dev_cli_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{
    "policyName": "DevCliTestThingPolicy",
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/DevCliTestThingPolicy",
    "policyDocument": "{\n    \"Version\": \"2012-10-17\",       \n    \"Statement\": [\n        {\n            \"Effect\": \"Allow\",\n            \"Action\": [\n                \"iot:Publish\",\n                \"iot:Subscribe\",\n    \"iot:Receive\",\n                \"iot:Connect\"\n            ],\n \"Resource\": [\n                \"*\"\n            ]\n        }\n    ]\n}\n",
    "policyVersionId": "1"
}
```

5. Run this command to attach the policy to the device certificate. Replace *certificateArn* with the `certificateArn` value you saved earlier.

```
aws iot attach-policy \
--policy-name "DevCliTestThingPolicy" \
--target "certificateArn"
```

If successful, this command returns nothing.

6. Run this command to attach the device certificate to the AWS IoT thing resource. Replace *certificateArn* with the `certificateArn` value you saved earlier.

```
aws iot attach-thing-principal \
--thing-name "DevCliTestThing" \
--principal "certificateArn"
```

If successful, this command returns nothing.

After you successfully provisioned your device in AWS IoT, you're ready to continue to the section called "Configure Device Client and test connectivity".

Configure the AWS IoT Device Client to test connectivity

The procedures in this section configure the AWS IoT Device Client to publish an MQTT message from your Raspberry Pi.

Procedures in this section:

- Create the config file
- Open MQTT test client
- Run AWS IoT Device Client

Create the config file

This procedure creates the config file to test the AWS IoT Device Client.

To create the config file to test the AWS IoT Device Client

- In the terminal window on your local host computer that's connected to your Raspberry Pi:

  a. Enter these commands to create a directory for the config files and set the permission on the directory:

  ```
  mkdir ~/dc-configs
  chmod 745 ~/dc-configs
  ```

  b. Open a text editor, such as nano.

  c. Copy this JSON document and paste it into your open text editor.

  ```
  {
    "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",
    "cert": "~/certs/testconn/device.pem.crt",
    "key": "~/certs/testconn/private.pem.key",
    "root-ca": "~/certs/AmazonRootCA1.pem",
    "thing-name": "DevCliTestThing",
    "logging": {
  ```

```
    "enable-sdk-logging": true,
    "level": "DEBUG",
    "type": "STDOUT",
    "file": ""
  },
  "jobs": {
    "enabled": false,
    "handler-directory": ""
  },
  "tunneling": {
    "enabled": false
  },
  "device-defender": {
    "enabled": false,
    "interval": 300
  },
  "fleet-provisioning": {
    "enabled": false,
    "template-name": "",
    "template-parameters": "",
    "csr-file": "",
    "device-key": ""
  },
  "samples": {
    "pub-sub": {
      "enabled": true,
      "publish-topic": "test/dc/pubtopic",
      "publish-file": "",
      "subscribe-topic": "test/dc/subtopic",
      "subscribe-file": ""
    }
  },
  "config-shadow": {
    "enabled": false
  },
  "sample-shadow": {
    "enabled": false,
    "shadow-name": "",
    "shadow-input-file": "",
    "shadow-output-file": ""
  }
}
```

d. Replace the *endpoint* value with device data endpoint for your AWS account that you found in the section called “Provision your device in AWS IoT Core”.

e. Save the file in your text editor as **~/dc-configs/dc-testconn-config.json**.

f. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-testconn-config.json
```

After you save the file, you're ready to continue to the section called “Open MQTT test client”.

Open MQTT test client

This procedure prepares the **MQTT test client** in the AWS IoT console to subscribe to the MQTT message that the AWS IoT Device Client publishes when it runs.

To prepare the MQTT test client to subscribe to all MQTT messages

1. On your local host computer, in the AWS IoT console, choose **MQTT test client**.
2. In the **Subscribe to a topic** tab, in **Topic filter**, enter **#** (a single pound sign), and choose **Subscribe** to subscribe to every MQTT topic.
3. Below the **Subscriptions** label, confirm that you see **#** (a single pound sign).

Leave the window with the **MQTT test client** open as you continue to the section called “Run AWS IoT Device Client”.

Run AWS IoT Device Client

This procedure runs the AWS IoT Device Client so that it publishes a single MQTT message that the **MQTT test client** receives and displays.

To send an MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window that's connected to your Raspberry Pi and the window with the **MQTT test client** are visible while you perform this procedure.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in the section called “Create the config file”.

```
cd ~/aws-iot-device-client/build
```

```
./aws-iot-device-client --config-file ~/dc-configs/dc-testconn-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

If no errors are displayed in the terminal window, review the **MQTT test client**.

3. In the **MQTT test client**, in the Subscriptions window, see the *Hello World!* message sent to the `test/dc/pubtopic` message topic.
4. If the AWS IoT Device Client displays no errors and you see *Hello World!* sent to the `test/dc/pubtopic` message in the **MQTT test client**, you've demonstrated a successful connection.
5. In the terminal window, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client is running correctly on your Raspberry Pi and can communicate with AWS IoT, you can continue to the the section called "Communicate with Device client using MQTT".

Tutorial: Demonstrate MQTT message communication with the AWS IoT Device Client

This tutorial demonstrates how the AWS IoT Device Client can subscribe to and publish MQTT messages, which are commonly used in IoT solutions.

To start this tutorial:

- Have your local host computer and Raspberry Pi configured as used in the previous section.

  If you saved the microSD card image after installing the AWS IoT Device Client, you can use a microSD card with that image with your Raspberry Pi.
- If you have run this demo before, review ??? to delete all AWS IoT resources that you created in earlier runs to avoid duplicate resource errors.

This tutorial takes about 45 minutes to complete.

When you're finished with this topic:

- You'll have demonstrated different ways that your IoT device can subscribe to MQTT messages from AWS IoT and publish MQTT messages to AWS IoT.

Required equipment:

- Your local development and testing environment from the previous section
- The Raspberry Pi that you used in the previous section
- The microSD memory card from the Raspberry Pi that you used in the previous section

Procedures in this tutorial

- Prepare the Raspberry Pi to demonstrate MQTT message communication
- Demonstrate publishing messages with the AWS IoT Device Client
- Demonstrate subscribing to messages with the AWS IoT Device Client

Prepare the Raspberry Pi to demonstrate MQTT message communication

This procedure creates the resources in AWS IoT and in the Raspberry Pi to demonstrate MQTT message communication using the AWS IoT Device Client.

Procedures in this section:

- Create the certificate files to demonstrate MQTT communication
- Provision your device to demonstrate MQTT communication
- Configure the AWS IoT Device Client config file and MQTT test client to demonstrate MQTT communication

Create the certificate files to demonstrate MQTT communication

This procedure creates the device certificate files for this demo.

To create and download the device certificate files for your Raspberry Pi

1. In the terminal window on your local host computer, enter the following command to create the device certificate files for your device.

```
mkdir ~/certs/pubsub
aws iot create-keys-and-certificate \
--set-as-active \
--certificate-pem-outfile "~/certs/pubsub/device.pem.crt" \
--public-key-outfile "~/certs/pubsub/public.pem.key" \
```

```
--private-key-outfile "~/certs/pubsub/private.pem.key"
```

The command returns a response like the following. Save the *certificateArn* value for later use.

```
{
"certificateArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fcd810f3aea2d92abc227d269",
"certificateId":
 "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fcd810f3aea2d92abc227d269",
"certificatePem": "-----BEGIN CERTIFICATE-----
\nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n-----END CERTIFICATE-----
\n",
"keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n-----END PUBLIC KEY-----
\n",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----
\nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n-----END RSA PRIVATE KEY-----\n"
}
}
```

2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 700 ~/certs/pubsub
chmod 644 ~/certs/pubsub/*
chmod 600 ~/certs/pubsub/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

```
ls -l ~/certs/pubsub
```

The output of the command should be the same as what you see here, except the file dates and times will be different.

```
-rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt
-rw------- 1 pi pi 1675 Oct 28 13:02 private.pem.key
-rw-r--r-- 1 pi pi  451 Oct 28 13:02 public.pem.key
```

4. Enter these commands to create the directories for the log files.

```
mkdir ~/.aws-iot-device-client
mkdir ~/.aws-iot-device-client/log
chmod 745 ~/.aws-iot-device-client/log
echo " " > ~/.aws-iot-device-client/log/aws-iot-device-client.log
echo " " > ~/.aws-iot-device-client/log/pubsub_rx_msgs.log
chmod 600 ~/.aws-iot-device-client/log/*
```

Provision your device to demonstrate MQTT communication

This section creates the AWS IoT resources that provision your Raspberry Pi in AWS IoT.

To provision your device in AWS IoT:

1. In the terminal window on your local host computer, enter the following command to get the address of the device data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type IoT:Data-ATS
   ```

   The endpoint value hasn't changed since the time you ran this command for the previous tutorial. Running the command again here is done to make it easy to find and paste the data endpoint value into the config file used in this tutorial.

   The command from the previous steps returns a response like the following. Record the *endpointAddress* value for later use.

   ```
   {
   "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. Enter this command to create a new AWS IoT thing resource for your Raspberry Pi.

   ```
   aws iot create-thing --thing-name "PubSubTestThing"
   ```

   Because an AWS IoT thing resource is a *virtual* representation of your device in the cloud, we can create multiple thing resources in AWS IoT to use for different purposes. They can all be used by the same physical IoT device to represent different aspects of the device.

These tutorials will only use one thing resource at a time to represent the Raspberry Pi. This way, in these tutorials, they represent the different demos so that after you create the AWS IoT resources for a demo, you can go back and repeat the demo using the resources you created specifically for each.

If your AWS IoT thing resource was created, the command returns a response like this.

```
{
"thingName": "PubSubTestThing",
"thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/PubSubTestThing",
"thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"
}
```

3. In the terminal window:

   a. Open a text editor, such as nano.

   b. Copy this JSON document and paste it into your open text editor.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:client/PubSubTestThing"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/test/dc/pubtopic"
            ]
        },
        {
            "Effect": "Allow",
```

```
                "Action": [
                    "iot:Subscribe"
                ],
                "Resource": [
                    "arn:aws:iot:us-west-2:123456789012:topicfilter/test/dc/
subtopic"
                ]
            },
            {
                "Effect": "Allow",
                "Action": [
                    "iot:Receive"
                ],
                "Resource": [
                    "arn:aws:iot:us-west-2:123456789012:topic/test/dc/subtopic"
                ]
            }
        ]
}
```

c. In the editor, in each `Resource` section of the policy document, replace *`us-west-2:57EXAMPLE833`* with your AWS Region, a colon character (:), and your 12-digit AWS account number.

d. Save the file in your text editor as **`~/policies/pubsub_test_thing_policy.json`**.

4. Run this command to use the policy document from the previous steps to create an AWS IoT policy.

```
aws iot create-policy \
--policy-name "PubSubTestThingPolicy" \
--policy-document "file://~/policies/pubsub_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{
                                    "policyName": "PubSubTestThingPolicy",
                                    "policyArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:policy/PubSubTestThingPolicy",
                                    "policyDocument": "{\n\"Version\":
 \"2012-10-17\",       \n\"Statement\": [\n{\n\"Effect\": \"Allow\",\n
\"Action\": [\n\"iot:Connect\"\n],\n\"Resource\": [\n\"arn:aws:iot:us-
west-2:57EXAMPLE833:client/PubSubTestThing\"\n]\n},\n{\n\"Effect\": \"Allow
```

```
\",\n\"Action\": [\n\"iot:Publish\"\n],\n\"Resource\": [\n\"arn:aws:iot:us-
west-2:57EXAMPLE833:topic/test/dc/pubtopic\"\n]\n},\n{\n\"Effect\": \"Allow
\",\n\"Action\": [\n\"iot:Subscribe\"\n],\n\"Resource\": [\n\"arn:aws:iot:us-
west-2:57EXAMPLE833:topicfilter/test/dc/subtopic\"\n]\n},\n{\n\"Effect\": \"Allow
\",\n\"Action\": [\n\"iot:Receive\"\n],\n\"Resource\": [\n\"arn:aws:iot:us-
west-2:57EXAMPLE833:topic/test/dc/*\"\n]\n}\n]\n}\n",
                                    "policyVersionId": "1"
                                    }
```

5. Run this command to attach the policy to the device certificate. Replace *`certificateArn`* with the `certificateArn` value you saved earlier in this section.

   ```
   aws iot attach-policy \
   --policy-name "PubSubTestThingPolicy" \
   --target "certificateArn"
   ```

   If successful, this command returns nothing.

6. Run this command to attach the device certificate to the AWS IoT thing resource. Replace *`certificateArn`* with the `certificateArn` value you saved earlier in this section.

   ```
   aws iot attach-thing-principal \
   --thing-name "PubSubTestThing" \
   --principal "certificateArn"
   ```

   If successful, this command returns nothing.

After you successfully provision your device in AWS IoT, you're ready to continue to [the section called "Configure Device Client config file and MQTT client"](#).

Configure the AWS IoT Device Client config file and MQTT test client to demonstrate MQTT communication

This procedure creates a config file to test the AWS IoT Device Client.

To create the config file to test the AWS IoT Device Client

1. In the terminal window on your local host computer that's connected to your Raspberry Pi:

   a. Open a text editor, such as `nano`.

   b. Copy this JSON document and paste it into your open text editor.

```
{
  "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",
  "cert": "~/certs/pubsub/device.pem.crt",
  "key": "~/certs/pubsub/private.pem.key",
  "root-ca": "~/certs/AmazonRootCA1.pem",
  "thing-name": "PubSubTestThing",
  "logging": {
    "enable-sdk-logging": true,
    "level": "DEBUG",
    "type": "STDOUT",
    "file": ""
  },
  "jobs": {
    "enabled": false,
    "handler-directory": ""
  },
  "tunneling": {
    "enabled": false
  },
  "device-defender": {
    "enabled": false,
    "interval": 300
  },
  "fleet-provisioning": {
    "enabled": false,
    "template-name": "",
    "template-parameters": "",
    "csr-file": "",
    "device-key": ""
  },
  "samples": {
    "pub-sub": {
      "enabled": true,
      "publish-topic": "test/dc/pubtopic",
      "publish-file": "",
      "subscribe-topic": "test/dc/subtopic",
      "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
    }
  },
  "config-shadow": {
    "enabled": false
  },
  "sample-shadow": {
```

```
    "enabled": false,
    "shadow-name": "",
    "shadow-input-file": "",
    "shadow-output-file": ""
  }
}
```

c. Replace the *endpoint* value with device data endpoint for your AWS account that you found in the section called “Provision your device in AWS IoT Core”.

d. Save the file in your text editor as **~/dc-configs/dc-pubsub-config.json**.

e. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-pubsub-config.json
```

2. To prepare the **MQTT test client** to subscribe to all MQTT messages:

   a. On your local host computer, in the AWS IoT console, choose **MQTT test client**.

   b. In the **Subscribe to a topic** tab, in **Topic filter**, enter **#** (a single pound sign), and choose **Subscribe**.

   c. Below the **Subscriptions** label, confirm that you see **#** (a single pound sign).

   Leave the window with the **MQTT test client** open while you continue this tutorial.

After you save the file and configure the **MQTT test client**, you're ready to continue to the section called “Publish messages with IoT Device Client”.

Demonstrate publishing messages with the AWS IoT Device Client

The procedures in this section demonstrate how the AWS IoT Device Client can send default and custom MQTT messages.

These policy statements in the policy that you created in the previous step for these exercises give the Raspberry Pi permission to perform these actions:

- **`iot:Connect`**

  Gives the client named `PubSubTestThing`, your Raspberry Pi running the AWS IoT Device Client, to connect.

```
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing"
      ]
    }
```

- **`iot:Publish`**

  Gives the Raspberry Pi permission to publish messages with an MQTT topic of `test/dc/pubtopic`.

```
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic"
      ]
    }
```

  The `iot:Publish` action gives permission to publish to the MQTT topics listed in the Resource array. The *content* of those messages is not controlled by the policy statement.

Publish the default message using the AWS IoT Device Client

This procedure runs the AWS IoT Device Client so that it publishes a single default MQTT message that the **MQTT test client** receives and displays.

To send the default MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window on your local host computer that's connected to your Raspberry Pi and the window with the **MQTT test client** are visible while you perform this procedure.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in the section called “Create the config file”.

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-config.json
```

In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

If no errors are displayed in the terminal window, review the **MQTT test client**.

3. In the **MQTT test client**, in the **Subscriptions** window, see the *Hello World!* message sent to the `test/dc/pubtopic` message topic.
4. If the AWS IoT Device Client displays no errors and you see *Hello World!* sent to the `test/dc/pubtopic` message in the **MQTT test client**, you've demonstrated a successful connection.
5. In the terminal window, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client published the default MQTT message, you can continue to the the section called “Publish custom MQTT message”.

Publish a custom message using the AWS IoT Device Client

The procedures in this section create a custom MQTT message and then runs the AWS IoT Device Client so that it publishes the custom MQTT message one time for the **MQTT test client** to receive and display.

Create a custom MQTT message for the AWS IoT Device Client

Perform these steps in the terminal window on the local host computer that's connected to your Raspberry Pi.

To create a custom message for the AWS IoT Device Client to publish

1. In the terminal window, open a text editor, such as `nano`.
2. Into the text editor, copy and paste the following JSON document. This will be the MQTT message payload that the AWS IoT Device Client publishes.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
```

```
    "velocity": 22,
    "bearing": 255
  }
}
```

3. Save the contents of the text editor as **~/messages/sample-ws-message.json**.
4. Enter the following command to set the permissions of the message file that you just created.

```
chmod 600 ~/messages/*
```

To create a config file for the AWS IoT Device Client to use to send the custom message

1. In the terminal window, in a text editor such as nano, open the existing AWS IoT Device Client config file: **~/dc-configs/dc-pubsub-config.json**.
2. Edit the `samples` object to look like this. No other part of this file needs to be changed.

```
  "samples": {
    "pub-sub": {
      "enabled": true,
      "publish-topic": "test/dc/pubtopic",
      "publish-file": "~/messages/sample-ws-message.json",
      "subscribe-topic": "test/dc/subtopic",
      "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
```

3. Save the contents of the text editor as **~/dc-configs/dc-pubsub-custom-config.json**.
4. Run this command to set the permissions on the new config file.

```
chmod 644 ~/dc-configs/dc-pubsub-custom-config.json
```

Publish the custom MQTT message by using the AWS IoT Device Client

This change affects only the *contents* of the MQTT message payload, so the current policy will continue to work. However, if the *MQTT topic* (as defined by the `publish-topic` value in `~/dc-configs/dc-pubsub-custom-config.json`) was changed, the `iot::Publish` policy statement would also need to be modified to allow the Raspberry Pi to publish to the new MQTT topic.

To send the MQTT message from the AWS IoT Device Client

1. Make sure that both the terminal window and the window with the **MQTT test client** are visible while you perform this procedure. Also, make sure that your **MQTT test client** is still subscribed to the **#** topic filter. If it isn't, subscribe to the **#** topic filter again.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in the section called "Create the config file".

   ```
   cd ~/aws-iot-device-client/build
   ./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-custom-config.json
   ```

   In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

   If no errors are displayed in the terminal window, review the MQTT test client.
3. In the **MQTT test client**, in the **Subscriptions** window, see the custom message payload sent to the `test/dc/pubtopic` message topic.
4. If the AWS IoT Device Client displays no errors and you see the custom message payload that you published to the `test/dc/pubtopic` message in the **MQTT test client**, you've published a custom message successfully.
5. In the terminal window, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.

After you've demonstrated that the AWS IoT Device Client published a custom message payload, you can continue to the section called "Subscribe to messages with IoT Device Client".

Demonstrate subscribing to messages with the AWS IoT Device Client

In this section, you'll demonstrate two types of message subscriptions:

- Single topic subscription
- Wild-card topic subscription

These policy statements in the policy created for these exercises give the Raspberry Pi permission to perform these actions:

- **`iot:Receive`**

Gives the AWS IoT Device Client permission to receive MQTT topics that match those named in the `Resource` object.

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Receive"
  ],
  "Resource": [
    "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/subtopic"
  ]
}
```

- **`iot:Subscribe`**

  Gives the AWS IoT Device Client permission to subscribe to MQTT topic filters that match those named in the `Resource` object.

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Subscribe"
  ],
  "Resource": [
    "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic"
  ]
}
```

Subscribe to a single MQTT message topic

This procedure demonstrates how the AWS IoT Device Client can subscribe to and log MQTT messages.

In the terminal window on your local host computer that's connected to your Raspberry Pi, list the contents of **`~/dc-configs/dc-pubsub-custom-config.json`** or open the file in a text editor to review its contents. Locate the `samples` object, which should look like this.

```
"samples": {
  "pub-sub": {
    "enabled": true,
```

```
    "publish-topic": "test/dc/pubtopic",
    "publish-file": "~/messages/sample-ws-message.json",
    "subscribe-topic": "test/dc/subtopic",
    "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
```

Notice the `subscribe-topic` value is the MQTT topic to which the AWS IoT Device Client will subscribe when it runs. The AWS IoT Device Client writes the message payloads that it receives from this subscription to the file named in the `subscribe-file` value.

To subscribe to a MQTT message topic from the AWS IoT Device Client

1. Make sure that both the terminal window and the window with the MQTT test client are visible while you perform this procedure. Also, make sure that your **MQTT test client** is still subscribed to the **#** topic filter. If it isn't, subscribe to the **#** topic filter again.
2. In the terminal window, enter these commands to run the AWS IoT Device Client using the config file created in the section called "Create the config file".

   ```
   cd ~/aws-iot-device-client/build
   ./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-custom-config.json
   ```

   In the terminal window, the AWS IoT Device Client displays information messages and any errors that occur when it runs.

   If no errors are displayed in the terminal window, continue in the AWS IoT console.
3. In the AWS IoT console, in the **MQTT test client**, choose the **Publish to a topic** tab.
4. In **Topic name**, enter **`test/dc/subtopic`**
5. In **Message payload**, review the message contents.
6. Choose **Publish** to publish the MQTT message.
7. In the terminal window, watch for the *message received* entry from the AWS IoT Device Client that looks like this.

   ```
   2021-11-10T16:02:20.890Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on
    subscribe topic, size: 45 bytes
   ```
8. After you see the *message received* entry that shows the message was received, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.
9. Enter this command to view the end of the message log file and see the message you published from the **MQTT test client**.

```
tail ~/.aws-iot-device-client/log/pubsub_rx_msgs.log
```

By viewing the message in the log file, you've demonstrated that the AWS IoT Device Client received the message that you published from the MQTT test client.

Subscribe to multiple MQTT message topic using wildcard characters

These procedures demonstrate how the AWS IoT Device Client can subscribe to and log MQTT messages using wildcard characters. To do this, you'll:

1. Update the topic filter that the AWS IoT Device Client uses to subscribe to MQTT topics.
2. Update the policy used by the device to allow the new subscriptions.
3. Run the AWS IoT Device Client and publish messages from the MQTT test console.

To create a config file to subscribe to multiple MQTT message topics by using a wildcard MQTT topic filter

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, open **`~/dc-configs/dc-pubsub-custom-config.json`** for editing and locate the `samples` object.
2. In the text editor, locate the `samples` object and update the `subscribe-topic` value to look like this.

   ```
   "samples": {
     "pub-sub": {
       "enabled": true,
       "publish-topic": "test/dc/pubtopic",
       "publish-file": "~/messages/sample-ws-message.json",
       "subscribe-topic": "test/dc/#",
       "subscribe-file": "~/.aws-iot-device-client/log/pubsub_rx_msgs.log"
   ```

   The new `subscribe-topic` value is an [MQTT topic filter](https://docs.aws.amazon.com/iot/latest/developerguide/topics.html#topicfilters) with an MQTT wild card character at the end. This describes a subscription to all MQTT topics that start with `test/dc/`. The AWS IoT Device Client writes the message payloads that it receives from this subscription to the file named in `subscribe-file`.

3. Save the modified config file as **~/dc-configs/dc-pubsub-wild-config.json**, and exit the editor.

To modify the policy used by your Raspberry Pi to allow subscribing to and receiving multiple MQTT message topics

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, in your favorite text editor, open **~/policies/pubsub_test_thing_policy.json** for editing, and then locate the `iot::Subscribe` and `iot::Receive` policy statements in the file.
2. In the `iot::Subscribe` policy statement, update the string in the Resource object to replace `subtopic` with `*`, so that it looks like this.

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Subscribe"
  ],
  "Resource": [
    "arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/*"
  ]
}
```

> **Note**
>
> The MQTT topic filter wild card characters are the + (plus sign) and the # (pound sign). A subscription request with a # at the end subscribes to all topics that start with the string that precedes the # character (for example, `test/dc/` in this case).
> The resource value in the policy statement that authorizes this subscription, however, must use a `*` (an asterisk) in place of the # (a pound sign) in the topic filter ARN. This is because the policy processor uses a different wild card character than MQTT uses.
> For more information about using wild card characters for topics and topic filters in policies, see Using wildcard characters in MQTT and AWS IoT Core policies.

3. In the `iot::Receive` policy statement, update the string in the Resource object to replace `subtopic` with `*`, so that it looks like this.

```
{
```

```
      "Effect": "Allow",
      "Action": [
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*"
      ]
    }
```

4. Save the updated policy document as **~/policies/pubsub_wild_test_thing_policy.json**, and exit the editor.

5. Enter this command to update the policy for this tutorial to use the new resource definitions.

```
aws iot create-policy-version \
--set-as-default \
--policy-name "PubSubTestThingPolicy" \
--policy-document "file://~/policies/pubsub_wild_test_thing_policy.json"
```

   If the command succeeds, it returns a response like this. Notice that `policyVersionId` is now 2, indicating this is the second version of this policy.

   If you successfully updated the policy, you can continue to the next procedure.


```
{
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/PubSubTestThingPolicy",
    "policyDocument": "{\n  \"Version\": \"2012-10-17\",       \n  \"Statement\": [\n    {\n      \"Effect\": \"Allow\",\n      \"Action\": [\n        \"iot:Connect\"\n      ],\n      \"Resource\": [\n        \"arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing\"\n      ]\n    },\n    {\n      \"Effect\": \"Allow\",\n      \"Action\": [\n        \"iot:Publish\"\n      ],\n      \"Resource\": [\n        \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic\"\n      ]\n    },\n    {\n      \"Effect\": \"Allow\",\n      \"Action\": [\n        \"iot:Subscribe\"\n      ],\n      \"Resource\": [\n        \"arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/*\"\n      ]\n    },\n    {\n      \"Effect\": \"Allow\",\n      \"Action\": [\n        \"iot:Receive\"\n      ],\n      \"Resource\": [\n        \"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*\"\n      ]\n    }\n  ]\n}\n",
    "policyVersionId": "2",
    "isDefaultVersion": true
}
```

If you get an error that there are too many policy versions to save a new one, enter this command to list the current versions of the policy. Review the list that this command returns to find a policy version that you can delete.

```
aws iot list-policy-versions --policy-name "PubSubTestThingPolicy"
```

Enter this command to delete a version that you no longer need. Note that you can't delete the default policy version. The default policy version is the one with a `isDefaultVersion` value of `true`.

```
aws iot delete-policy-version \
--policy-name "PubSubTestThingPolicy" \
--policy-version-id policyId
```

After deleting a policy version, retry this step.

With the updated config file and policy, you're ready to demonstrate wild card subscriptions with the AWS IoT Device Client.

To demonstrate how the AWS IoT Device Client subscribes to and receives multiple MQTT message topics

1. In the **MQTT test client**, check the subscriptions. If the **MQTT test client** is subscribed to the to the in the **#** topic filter, continue to the next step. If not, in the **MQTT test client**, in **Subscribe to a topic** tab, in **Topic filter**, enter **#** (a pound sign character), and then choose **Subscribe** to subscribe to it.
2. In the terminal window on your local host computer that's connected to your Raspberry Pi, enter these commands to start the AWS IoT Device Client.

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-pubsub-wild-config.json
```

3. While watching the AWS IoT Device Client output in the terminal window on the local host computer, return to the **MQTT test client**. In the **Publish to a topic** tab, in **Topic name**, enter **`test/dc/subtopic`** , and then choose **Publish**.
4. In the terminal window, confirm that the message was received by looking for a message such as:

```
2021-11-10T16:34:20.101Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on
 subscribe topic, size: 76 bytes
```

5. While watching the AWS IoT Device Client output in the terminal window of the local host computer, return to the **MQTT test client**. In the **Publish to a topic** tab, in **Topic name**, enter **`test/dc/subtopic2`** , and then choose **Publish**.

6. In the terminal window, confirm that the message was received by looking for a message such as:

```
2021-11-10T16:34:32.078Z [DEBUG] {samples/PubSubFeature.cpp}: Message received on
 subscribe topic, size: 77 bytes
```

7. After you see the messages that confirm both messages were received, enter **^C** (Ctrl-C) to stop the AWS IoT Device Client.

8. Enter this command to view the end of the message log file and see the message you published from the **MQTT test client**.

```
tail -n 20 ~/.aws-iot-device-client/log/pubsub_rx_msgs.log
```

> **Note**
>
> The log file contains only message payloads. The message topics are not recorded in the received message log file.
> You might also see the message published by the AWS IoT Device Client in the received log. This is because the wild card topic filter includes that message topic and, sometimes, the subscription request can be processed by message broker before the published message is sent to subscribers.

The entries in the log file demonstrate that the messages were received. You can repeat this procedure using other topic names. All messages that have a topic name that begins with `test/dc/` should be received and logged. Messages with topic names that begin with any other text are ignored.

After demonstrating how the AWS IoT Device Client can publish and subscribe to MQTT messages, continue to Tutorial: Demonstrate remote actions (jobs) with the AWS IoT Device Client.

Tutorial: Demonstrate remote actions (jobs) with the AWS IoT Device Client

In these tutorials, you'll configure and deploy jobs to your Raspberry Pi to demonstrate how you can send remote operations to your IoT devices.

To start this tutorial:

- Have your local host computer an Raspberry Pi configured as used in the previous section.
- If you haven't completed the tutorial in the previous section, you can try this tutorial by using the Raspberry Pi with a microSD card that has the image you saved after you installed the AWS IoT Device Client in (Optional) Save the microSD card image.
- If you have run this demo before, review ??? to delete all AWS IoT resources that you created in earlier runs to avoid duplicate resource errors.

This tutorial takes about 45 minutes to complete.

When you're finished with this topic:

- You'll have demonstrated different ways that your IoT device can use the AWS IoT Core to run remote operations that are managed by AWS IoT .

Required equipment:

- Your local development and testing environment that you tested in a previous section
- The Raspberry Pi that you tested in a previous section
- The microSD memory card from the Raspberry Pi that you tested in a previous section

Procedures in this tutorial

- Prepare the Raspberry Pi to run jobs
- Create and run the job in AWS IoT with AWS IoT Device Client

Prepare the Raspberry Pi to run jobs

The procedures in this section describe how to prepare your Raspberry Pi to run jobs by using the AWS IoT Device Client.

> **Note**
>
> These procedures are device specific. If you want to perform the procedures in this section with more than one device at the same time, each device will need its own policy and unique, device-specific certificate and thing name. To give each device its unique resources, perform this procedure one time for each device while changing the device-specific elements as described in the procedures.

Procedures in this tutorial

- Provision your Raspberry Pi to demonstrate jobs
- Configure the AWS IoT Device Client to run the jobs agent

Provision your Raspberry Pi to demonstrate jobs

The procedures in this section provision your Raspberry Pi in AWS IoT by creating AWS IoT resources and device certificates for it.

Topics

- Create and download device certificate files to demonstrate AWS IoT jobs
- Create AWS IoT resources to demonstrate AWS IoT jobs

Create and download device certificate files to demonstrate AWS IoT jobs

This procedure creates the device certificate files for this demo.

If you are preparing more than one device, this procedure must be performed on each device.

To create and download the device certificate files for your Raspberry Pi:

In the terminal window on your local host computer that's connected to your Raspberry Pi, enter these commands.

1. Enter the following command to create the device certificate files for your device.

   ```
   aws iot create-keys-and-certificate \
   --set-as-active \
   ```

```
--certificate-pem-outfile "~/certs/jobs/device.pem.crt" \
--public-key-outfile "~/certs/jobs/public.pem.key" \
--private-key-outfile "~/certs/jobs/private.pem.key"
```

The command returns a response like the following. Save the *certificateArn* value for later use.


```
{
"certificateArn": "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/76e7e4edb3e52f52334be2f387a06145b2aa4c7fcd810f3aea2d92abc227d269",
"certificateId":
 "76e7e4edb3e52f5233EXAMPLE7a06145b2aa4c7fcd810f3aea2d92abc227d269",
"certificatePem": "-----BEGIN CERTIFICATE-----
\nMIIDWTCCAkGgAwIBAgI_SHORTENED_FOR_EXAMPLE_Lgn4jfgtS\n-----END CERTIFICATE-----
\n",
"keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BA_SHORTENED_FOR_EXAMPLE_ImwIDAQAB\n-----END PUBLIC KEY-----
\n",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----
\nMIIEowIBAAKCAQE_SHORTENED_FOR_EXAMPLE_T9RoDiukY\n-----END RSA PRIVATE KEY-----\n"
}
}
```


2. Enter the following commands to set the permissions on the certificate directory and its files.

```
chmod 700 ~/certs/jobs
chmod 644 ~/certs/jobs/*
chmod 600 ~/certs/jobs/private.pem.key
```

3. Run this command to review the permissions on your certificate directories and files.

```
ls -l ~/certs/jobs
```

The output of the command should be the same as what you see here, except the file dates and times will be different.

```
-rw-r--r-- 1 pi pi 1220 Oct 28 13:02 device.pem.crt
-rw------- 1 pi pi 1675 Oct 28 13:02 private.pem.key
-rw-r--r-- 1 pi pi  451 Oct 28 13:02 public.pem.key
```

After you have downloaded the device certificate files to your Raspberry Pi, you're ready to continue to the section called “Provision Raspberry Pi for jobs”.

Create AWS IoT resources to demonstrate AWS IoT jobs

Create the AWS IoT resources for this device.

If you are preparing more than one device, this procedure must be performed for each device.

To provision your device in AWS IoT:

In the terminal window on your local host computer that's connected to your Raspberry Pi:

1. Enter the following command to get the address of the device data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type IoT:Data-ATS
   ```

   The endpoint value hasn’t changed since the last time you ran this command. Running the command again here makes it easy to find and paste the data endpoint value into the config file used in this tutorial.

   The **describe-endpoint** command returns a response like the following. Record the *endpointAddress* value for later use.

   ```
   {
   "endpointAddress": "a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. Replace *uniqueThingName* with a unique name for your device. If you want to perform this tutorial with multiple devices, give each device its own name. For example, **TestDevice01**, **TestDevice02**, and so on.

   Enter this command to create a new AWS IoT thing resource for your Raspberry Pi.

   ```
   aws iot create-thing --thing-name "uniqueThingName"
   ```

   Because an AWS IoT thing resource is a *virtual* representation of your device in the cloud, we can create multiple thing resources in AWS IoT to use for different purposes. They can all be used by the same physical IoT device to represent different aspects of the device.

Note

When you want to secure the policy for multiple devices, you can use `${iot:Thing.ThingName}` instead of the static thing name, *uniqueThingName*.

These tutorials will only use one thing resource at a time per device. This way, in these tutorials, they represent the different demos so that after you create the AWS IoT resources for a demo, you can go back and repeat the demos using the resources you created specifically for each.

If your AWS IoT thing resource was created, the command returns a response like this. Record the *thingArn* value for use later when you create the job to run on this device.

```
{
"thingName": "uniqueThingName",
"thingArn": "arn:aws:iot:us-west-2:57EXAMPLE833:thing/uniqueThingName",
"thingId": "8ea78707-32c3-4f8a-9232-14bEXAMPLEfd"
}
```

3. In the terminal window:

   a. Open a text editor, such as nano.

   b. Copy this JSON document and paste it into your open text editor.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:client/uniqueThingName"
            ]
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/test/dc/pubtopic",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/events/job/
*",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/events/
jobExecution/*",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topicfilter/test/dc/
subtopic",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/events/
jobExecution/*",
                "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/test/dc/subtopic",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:DescribeJobExecution",
                "iot:GetPendingJobExecutions",
                "iot:StartNextPendingJobExecution",
```

```
                "iot:UpdateJobExecution"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName"
            ]
        }
    ]
}
```

c. In the editor, in the `Resource` section of every policy statement, replace *`us-west-2:57EXAMPLE833`* with your AWS Region, a colon character (:), and your 12-digit AWS account number.

d. In the editor, in every policy statement, replace *`uniqueThingName`* with the thing name you gave this thing resource.

e. Save the file in your text editor as **`~/policies/jobs_test_thing_policy.json`**.

   If you are running this procedure for multiple devices, save the file to this file name on each device.

4. Replace *`uniqueThingName`* with the thing name for the device, and then run this command to create an AWS IoT policy that is tailored for that device.

```
aws iot create-policy \
--policy-name "JobTestPolicyForuniqueThingName" \
--policy-document "file://~/policies/jobs_test_thing_policy.json"
```

If the policy is created, the command returns a response like this.

```
{
    "policyName": "JobTestPolicyForuniqueThingName",
    "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/
JobTestPolicyForuniqueThingName",
    "policyDocument": "{\n\"Version\": \"2012-10-17\",\n\"Statement\": [\n{\n
\"Effect\": \"Allow\",\n\"Action\": [\n\"iot:Connect\"\n],\n\"Resource\":
 [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:client/PubSubTestThing\"\n]\n},\n{\n
\"Effect\": \"Allow\",\n\"Action\": [\n\"iot:Publish\"\n],\n\"Resource\":
 [\n\"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/pubtopic\"\n]\n},\n{\n
\"Effect\": \"Allow\",\n\"Action\": [\n\"iot:Subscribe\"\n],\n\"Resource\": [\n
\"arn:aws:iot:us-west-2:57EXAMPLE833:topicfilter/test/dc/subtopic\"\n]\n},\n{\n
```

```
\"Effect\": \"Allow\",\n\"Action\": [\n\"iot:Receive\"\n],\n\"Resource\": [\n
\"arn:aws:iot:us-west-2:57EXAMPLE833:topic/test/dc/*\"\n]\n}\n]\n}\n",
    "policyVersionId": "1"
}
```

5. Replace *uniqueThingName* with the thing name for the device and *certificateArn* with the `certificateArn` value you saved earlier in this section for this device, and then run this command to attach the policy to the device certificate.

   ```
   aws iot attach-policy \
   --policy-name "JobTestPolicyForuniqueThingName" \
   --target "certificateArn"
   ```

   If successful, this command returns nothing.

6. Replace *uniqueThingName* with the thing name for the device, replace *certificateArn* with the `certificateArn` value that you saved earlier in this section, and then run this command to attach the device certificate to the AWS IoT thing resource.

   ```
   aws iot attach-thing-principal \
   --thing-name "uniqueThingName" \
   --principal "certificateArn"
   ```

   If successful, this command returns nothing.

After you successfully provisioned your Raspberry Pi, you're ready to repeat this section for another Raspberry Pi in your test or, if all devices have been provisioned, continue to the section called "Configure Device Client to run jobs".

Configure the AWS IoT Device Client to run the jobs agent

This procedure creates a config file for the AWS IoT Device Client to run the jobs agent:.

Note: if you are preparing more than one device, this procedure must be performed on each device.

To create the config file to test the AWS IoT Device Client:

1. In the terminal window on your local host computer that's connected to your Raspberry Pi:

   a. Open a text editor, such as `nano`.

   b. Copy this JSON document and paste it into your open text editor.

```
{
  "endpoint": "a3qEXAMPLEaffp-ats.iot.us-west-2.amazonaws.com",
  "cert": "~/certs/jobs/device.pem.crt",
  "key": "~/certs/jobs/private.pem.key",
  "root-ca": "~/certs/AmazonRootCA1.pem",
  "thing-name": "uniqueThingName",
  "logging": {
    "enable-sdk-logging": true,
    "level": "DEBUG",
    "type": "STDOUT",
    "file": ""
  },
  "jobs": {
    "enabled": true,
    "handler-directory": ""
  },
  "tunneling": {
    "enabled": false
  },
  "device-defender": {
    "enabled": false,
    "interval": 300
  },
  "fleet-provisioning": {
    "enabled": false,
    "template-name": "",
    "template-parameters": "",
    "csr-file": "",
    "device-key": ""
  },
  "samples": {
    "pub-sub": {
      "enabled": false,
      "publish-topic": "",
      "publish-file": "",
      "subscribe-topic": "",
      "subscribe-file": ""
    }
  },
  "config-shadow": {
    "enabled": false
  },
  "sample-shadow": {
```

```
    "enabled": false,
    "shadow-name": "",
    "shadow-input-file": "",
    "shadow-output-file": ""
  }
}
```

c. Replace the *endpoint* value with device data endpoint value for your AWS account that you found in the section called "Provision your device in AWS IoT Core".

d. Replace *uniqueThingName* with the thing name that you used for this device.

e. Save the file in your text editor as **~/dc-configs/dc-jobs-config.json**.

2. Run this command to set the file permissions of the new config file.

```
chmod 644 ~/dc-configs/dc-jobs-config.json
```

You won't use the **MQTT test client** for this test. While the device will exchange jobs-related MQTT messages with AWS IoT, job progress messages are only exchanged with the device running the job. Because job progress messages are only exchanged with the device running the job, you can't subscribe to them from another device, such as the AWS IoT console.

After you save the config file, you're ready to continue to the section called "Create and run IoT job".

Create and run the job in AWS IoT with AWS IoT Device Client

The procedures in this section create a job document and an AWS IoT job resource. After you create the job resource, AWS IoT sends the job document to the specified job targets on which a jobs agent applies the job document to the device or client.

Procedures in this section

- Create and store the job document for the IoT job
- Run a job in AWS IoT for one IoT device

Create and store the job document for the IoT job

This procedure creates a simple job document to include in an AWS IoT job resource. This job document displays "Hello world!" on the job target.

To create and store a job document:

1. Select the Amazon S3 bucket into which you'll save your job document. If you don't have an existing Amazon S3 bucket to use for this, you'll need to create one. For information about how to create Amazon S3 buckets, see the topics in Getting started with Amazon S3.
2. Create and save the job document for this job
   a. On your local host computer, open a text editor.
   b. Copy and paste this text into the editor.

      ```
      {
          "operation": "echo",
          "args": ["Hello world!"]
      }
      ```
   c. On the local host computer, save the contents of the editor to a file named **`hello-world-job.json`**.
   d. Confirm the file was saved correctly. Some text editors automatically append `.txt` to the file name when they save a text file. If your editor appended `.txt` to the file name, correct the file name before proceeding.
3. Replace the *`path_to_file`* with the path to **`hello-world-job.json`**, if it's not in your current directory, replace *`s3_bucket_name`* with the Amazon S3 bucket path to the bucket you selected, and then run this command to put your job document into the Amazon S3 bucket.

   ```
   aws s3api put-object \
   --key hello-world-job.json \
   --body path_to_file/hello-world-job.json --bucket s3_bucket_name
   ```

   The job document URL that identifies the job document that you stored in Amazon S3 is determined by replacing the *`s3_bucket_name`* and *`AWS_region`* in the following URL. Record the resulting URL to use later as the *`job_document_path`*

   ```
   https://s3_bucket_name.s3.AWS_Region.amazonaws.com/hello-world-job.json
   ```

> **Note**
>
> AWS security prevents you from being able to open this URL outside of your AWS account, for example by using a browser. The URL is used by the AWS IoT jobs engine, which has access to the file, by default. In a production environment, you'll need to make sure that your AWS IoT services have permission to access to the job documents stored in Amazon S3.

After you have saved the job document's URL, continue to the section called “Run job for single device”.

Run a job in AWS IoT for one IoT device

The procedures in this section start the AWS IoT Device Client on your Raspberry Pi to run the jobs agent on the device to wait for jobs to run. It also creates a job resource in AWS IoT, which will send the job to and run on your IoT device.

> **Note**
>
> This procedure runs a job on only a single device.

To start the jobs agent on your Raspberry Pi:

1. In the terminal window on your local host computer that's connected to your Raspberry Pi, run this command to start the AWS IoT Device Client.

```
cd ~/aws-iot-device-client/build
./aws-iot-device-client --config-file ~/dc-configs/dc-jobs-config.json
```

2. In the terminal window, confirm that the AWS IoT Device Client and displays these messages

```
2021-11-15T18:45:56.708Z [INFO]  {Main.cpp}: Jobs is enabled
                      .
                      .
                      .
2021-11-15T18:45:56.708Z [INFO]  {Main.cpp}: Client base has been notified that
 Jobs has started
```

```
2021-11-15T18:45:56.708Z [INFO]  {JobsFeature.cpp}: Running Jobs!
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to
 startNextPendingJobExecution accepted and rejected
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to
 nextJobChanged events
2021-11-15T18:45:56.708Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to
 updateJobExecutionStatusAccepted for jobId +
2021-11-15T18:45:56.738Z [DEBUG] {JobsFeature.cpp}: Ack received for
 SubscribeToUpdateJobExecutionAccepted with code {0}
2021-11-15T18:45:56.739Z [DEBUG] {JobsFeature.cpp}: Attempting to subscribe to
 updateJobExecutionStatusRejected for jobId +
2021-11-15T18:45:56.753Z [DEBUG] {JobsFeature.cpp}: Ack received for
 SubscribeToNextJobChanged with code {0}
2021-11-15T18:45:56.760Z [DEBUG] {JobsFeature.cpp}: Ack received for
 SubscribeToStartNextJobRejected with code {0}
2021-11-15T18:45:56.776Z [DEBUG] {JobsFeature.cpp}: Ack received for
 SubscribeToStartNextJobAccepted with code {0}
2021-11-15T18:45:56.776Z [DEBUG] {JobsFeature.cpp}: Ack received for
 SubscribeToUpdateJobExecutionRejected with code {0}
2021-11-15T18:45:56.777Z [DEBUG] {JobsFeature.cpp}: Publishing
 startNextPendingJobExecutionRequest
2021-11-15T18:45:56.785Z [DEBUG] {JobsFeature.cpp}: Ack received for
 StartNextPendingJobPub with code {0}
2021-11-15T18:45:56.785Z [INFO]  {JobsFeature.cpp}: No pending jobs are scheduled,
 waiting for the next incoming job
```

3. In the terminal window, after you see this message, continue to the next procedure and create the job resource. Note that it might not be the last entry in the list.

```
2021-11-15T18:45:56.785Z [INFO]  {JobsFeature.cpp}: No pending jobs are scheduled,
 waiting for the next incoming job
```

To create an AWS IoT job resource

1. On your local host computer:

   a. Replace *job_document_url* with the job document URL from the section called "Create and store job document".

   b. Replace *thing_arn* with the ARN of the thing resource you created for your device and then run this command.

```
aws iot create-job \
--job-id hello-world-job-1 \
--document-source "job_document_url" \
--targets "thing_arn" \
--target-selection SNAPSHOT
```

If successful, the command returns a result like this one.

```
{
  "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-job-1",
  "jobId": "hello-world-job-1"
}
```

2. In the terminal window, you should see output from the AWS IoT Device Client like this.

```
2021-11-15T18:02:26.688Z [INFO]  {JobsFeature.cpp}: No pending jobs are scheduled,
 waiting for the next incoming job
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Job ids differ
2021-11-15T18:10:24.890Z [INFO]  {JobsFeature.cpp}: Executing job: hello-world-
job-1
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Attempting to update job
 execution status!
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Not including stdout with the
 status details
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Not including stderr with the
 status details
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Assuming executable is in PATH
2021-11-15T18:10:24.890Z [INFO]  {JobsFeature.cpp}: About to execute: echo Hello
 world!
2021-11-15T18:10:24.890Z [DEBUG] {Retry.cpp}: Retryable function starting, it will
 retry until success
2021-11-15T18:10:24.890Z [DEBUG] {JobsFeature.cpp}: Created EphermalPromise for
 ClientToken 3TEWba9Xj6 in the updateJobExecution promises map
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Child process now running
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Child process about to call
 execvp
2021-11-15T18:10:24.890Z [DEBUG] {JobEngine.cpp}: Parent process now running, child
 PID is 16737
2021-11-15T18:10:24.891Z [DEBUG] {16737}: Hello world!
2021-11-15T18:10:24.891Z [DEBUG] {JobEngine.cpp}: JobEngine finished waiting for
 child process, returning 0
2021-11-15T18:10:24.891Z [INFO]  {JobsFeature.cpp}: Job exited with status: 0
```

```
2021-11-15T18:10:24.891Z [INFO]  {JobsFeature.cpp}: Job executed successfully!
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Attempting to update job
 execution status!
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Not including stdout with the
 status details
2021-11-15T18:10:24.891Z [DEBUG] {JobsFeature.cpp}: Not including stderr with the
 status details
2021-11-15T18:10:24.892Z [DEBUG] {Retry.cpp}: Retryable function starting, it will
 retry until success
2021-11-15T18:10:24.892Z [DEBUG] {JobsFeature.cpp}: Created EphermalPromise for
 ClientToken GmQ0HTzWGg in the updateJobExecution promises map
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Ack received for
 PublishUpdateJobExecutionStatus with code {0}
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Removing ClientToken 3TEWba9Xj6
 from the updateJobExecution promises map
2021-11-15T18:10:24.905Z [DEBUG] {JobsFeature.cpp}: Success response after
 UpdateJobExecution for job hello-world-job-1
2021-11-15T18:10:24.917Z [DEBUG] {JobsFeature.cpp}: Ack received for
 PublishUpdateJobExecutionStatus with code {0}
2021-11-15T18:10:24.918Z [DEBUG] {JobsFeature.cpp}: Removing ClientToken GmQ0HTzWGg
 from the updateJobExecution promises map
2021-11-15T18:10:24.918Z [DEBUG] {JobsFeature.cpp}: Success response after
 UpdateJobExecution for job hello-world-job-1
2021-11-15T18:10:25.861Z [INFO]  {JobsFeature.cpp}: No pending jobs are scheduled,
 waiting for the next incoming job
```

3. While the AWS IoT Device Client is running and waiting for a job, you can submit another job by changing the `job-id` value and re-running the **create-job** from Step 1.

When you're done running jobs, in the terminal window, enter **^C** (control-C) to stop the AWS IoT Device Client.

Tutorial: Cleaning up after running the AWS IoT Device Client tutorials

The procedures in this tutorial walk you through removing the files and resources you created while completing the tutorials in this learning path.

Procedures in this tutorial

- Step 1: Cleaning up your devices after building demos with the AWS IoT Device Client
- Step 2: Cleaning up your AWS account after building demos with the AWS IoT Device Client

Step 1: Cleaning up your devices after building demos with the AWS IoT Device Client

This tutorial describes two options for how to clean up the microSD card after you built the demos in this learning path. Choose the option that provides the level of security that you need.

Note that cleaning the device's microSD card does not remove any AWS IoT resources that you created. To clean up the AWS IoT resources after you clean the device's microSD card, you should review the tutorial on the section called “Cleaning up after building demos with the AWS IoT Device Client”.

Option 1: Cleaning up by rewriting the microSD card

The easiest and most thorough way to clean the microSD card after completing the tutorials in this learning path is to overwrite the microSD card with a saved image file that you created while preparing your device the first time.

This procedure uses the local host computer to write a saved microSD card image to a microSD card.

> **Note**
>
> If your device doesn't use a removable storage medium for its operating system, refer to the procedure for that device.

To write a new image to the microSD card

1. On your local host computer, locate the saved microSD card image that you want to write to your microSD card.
2. Insert your microSD card into the local host computer.
3. Using an SD card imaging tool, write selected image file to the microSD card.
4. After writing the Raspberry Pi OS image to the microSD card, eject the microSD card and safely remove it from the local host computer.

Your microSD card is ready to use.

Option 2: Cleaning up by deleting user directories

To clean the microSD card after completing the tutorials without rewriting the microSD card image, you can delete the user directories individually. This is not as thorough as rewriting the microSD card from a saved image because it does not remove any system files that might have been installed.

If removing the user directories is sufficiently thorough for you needs, you can follow this procedure.

To delete this learning path's user directories from your device

1. Run these commands to delete the user directories, subdirectories, and all their files that were created in this learning path, in the terminal window connected to your device.

    > **Note**
    >
    > After you delete these directories and files, you won't be able to run the demos without completing the tutorials again.

    ```
    rm -Rf ~/dc-configs
    rm -Rf ~/policies
    rm -Rf ~/messages
    rm -Rf ~/certs
    rm -Rf ~/.aws-iot-device-client
    ```

2. Run these commands to delete the application source directories and files, in the terminal window connected to your device.

    > **Note**
    >
    > These commands don't uninstall any programs. They only remove the source files used to build and install them. After you delete these files, the AWS CLI and the AWS IoT Device Client might not work.

    ```
    rm -Rf ~/aws-cli
    rm -Rf ~/aws
    ```

```
rm -Rf ~/aws-iot-device-client
```

Step 2: Cleaning up your AWS account after building demos with the AWS IoT Device Client

These procedures help you identify and remove the AWS resources that you created while completing the tutorials in this learning path.

Clean up AWS IoT resources

This procedure helps you identify and remove the AWS IoT resources that you created while completing the tutorials in this learning path.

AWS IoT resources created in this learning path

Tutorial	Thing resource	Policy resource
the section called "Installing and configuring IoT device client"	**DevCliTestThing**	**DevCliTestThingPolicy**
the section called "Communicate with Device client using MQTT"	**PubSubTestThing**	**PubSubTestThingPolicy**
the section called "Run IoT jobs with the Device Client"	*user defined* (there could be more than one)	*user defined* (there could be more than one)

To delete the AWS IoT resources, follow this procedure for each thing resource that you created

1. Replace *thing_name* with the name of the thing resource you want to delete, and then run this command to list the certificates attached to the thing resource, from the local host computer.

   ```
   aws iot list-thing-principals --thing-name thing_name
   ```

   This command returns a response like this one that lists the certificates that are attached to *thing_name*. In most cases, there will only be one certificate in the list.

```
{
    "principals": [
        "arn:aws:iot:us-
west-2:57EXAMPLE833:cert/23853eea3cf0edc7f8a69c74abeafa27b2b52823cab5b3e156295e94b26ae8ac"
    ]
}
```

2. For each certificate listed by the previous command:

   a. Replace *certificate_ID* with the certificate ID from the previous command. The certificate ID is the alphanumeric characters that follow `cert/` in the ARN returned by the previous command. Then run this command to inactivate the certificate.

   ```
   aws iot update-certificate --new-status INACTIVE --certificate-
   id certificate_ID
   ```

   If successful, this command doesn't return anything.

   b. Replace *certificate_ARN* with the certificate ARN from the list of certificates returned earlier, and then run this command to list the policies attached to this certificate.

   ```
   aws iot list-attached-policies --target certificate_ARN
   ```

   This command returns a response like this one that lists the policies attached to the certificate. In most cases, there will only be one policy in the list.

   ```
   {
       "policies": [
           {
               "policyName": "DevCliTestThingPolicy",
               "policyArn": "arn:aws:iot:us-west-2:57EXAMPLE833:policy/
   DevCliTestThingPolicy"
           }
       ]
   }
   ```

   c. For each policy attached to the certificate:

i. Replace *policy_name* with the `policyName` value from the previous command, replace *certificate_ARN* with the certificate's ARN, and then run this command to detach the policy from the certificate.

```
aws iot detach-policy --policy-name policy_name --target certificate_ARN
```

If successful, this command doesn't return anything.

ii. Replace *policy_name* with the `policyName` value, and then run this command to see if the policy is attached to any more certificates.

```
aws iot list-targets-for-policy --policy-name policy_name
```

If the command returns an empty list like this, the policy is not attached to any certificates and you continue to list the policy versions. If there are still certificates attached to the policy, continue with the **detach-thing-principal** step.

```
{
    "targets": []
}
```

iii. Replace *policy_name* with the `policyName` value, and then run this command to check for policy versions. To delete the policy, it must have only one version.

```
aws iot list-policy-versions --policy-name policy_name
```

If the policy has only one version, like this example, you can skip to the **delete-policy** step and delete the policy now.

```
{
    "policyVersions": [
        {
            "versionId": "1",
            "isDefaultVersion": true,
            "createDate": "2021-11-18T01:02:46.778000+00:00"
        }
    ]
}
```

If the policy has more than one version, like this example, the policy versions with an `isDefaultVersion` value of `false` must be deleted before the policy can be deleted.

```
{
    "policyVersions": [
        {
            "versionId": "2",
            "isDefaultVersion": true,
            "createDate": "2021-11-18T01:52:04.423000+00:00"
        },
        {
            "versionId": "1",
            "isDefaultVersion": false,
            "createDate": "2021-11-18T01:30:18.083000+00:00"
        }
    ]
}
```

If you need to delete a policy version, replace *policy_name* with the `policyName` value, replace *version_ID* with the `versionId` value from the previous command, and then run this command to delete a policy version.

```
aws iot delete-policy-version --policy-name policy_name --policy-version-id version_ID
```

If successful, this command doesn't return anything.

After you delete a policy version, repeat this step until the policy has only one policy version.

iv. Replace *policy_name* with the `policyName` value, and then run this command to delete the policy.

```
aws iot delete-policy --policy-name policy_name
```

d. Replace *thing_name* with the thing's name, replace *certificate_ARN* with the certificate's ARN, and then run this command to detach the certificate from the thing resource.

```
aws iot detach-thing-principal --thing-name thing_name --
principal certificate_ARN
```

If successful, this command doesn't return anything.

e. Replace `certificate_ID` with the certificate ID from the previous command. The certificate ID is the alphanumeric characters that follow `cert/` in the ARN returned by the previous command. Then run this command to delete the certificate resource.

```
aws iot delete-certificate --certificate-id certificate_ID
```

If successful, this command doesn't return anything.

3. Replace `thing_name` with the thing's name, and then run this command to delete the thing.

```
aws iot delete-thing --thing-name thing_name
```

If successful, this command doesn't return anything.

Clean up AWS resources

This procedure helps you identify and remove other AWS resources that you created while completing the tutorials in this learning path.

Other AWS resources created in this learning path

Tutorial	Resource type	Resource name or ID
the section called “Run IoT jobs with the Device Client”	Amazon S3 object	hello-world-job.json
the section called “Run IoT jobs with the Device Client”	AWS IoT job resources	*user defined*

To delete the AWS resources created in this learning path

1. To delete the jobs created in this learning path

   a. Run this command to list the jobs in your AWS account.

```
aws iot list-jobs
```

The command returns a list of the AWS IoT jobs in your AWS account and AWS Region that looks like this.


```
{
    "jobs": [
        {
            "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-
job-2",
            "jobId": "hello-world-job-2",
            "targetSelection": "SNAPSHOT",
            "status": "COMPLETED",
            "createdAt": "2021-11-16T23:40:36.825000+00:00",
            "lastUpdatedAt": "2021-11-16T23:40:41.375000+00:00",
            "completedAt": "2021-11-16T23:40:41.375000+00:00"
        },
        {
            "jobArn": "arn:aws:iot:us-west-2:57EXAMPLE833:job/hello-world-
job-1",
            "jobId": "hello-world-job-1",
            "targetSelection": "SNAPSHOT",
            "status": "COMPLETED",
            "createdAt": "2021-11-16T23:35:26.381000+00:00",
            "lastUpdatedAt": "2021-11-16T23:35:29.239000+00:00",
            "completedAt": "2021-11-16T23:35:29.239000+00:00"
        }
    ]
}
```


b. For each job that you recognize from the list as a job you created in this learning path, replace *`jobId`* with the `jobId` value of the job to delete, and then run this command to delete an AWS IoT job.

```
aws iot delete-job --job-id jobId
```

If the command is successful, it returns nothing.

2. To delete the job documents you stored in an Amazon S3 bucket in this learning path.

a. Replace *bucket* with the name of the bucket you used, and then run this command to list the objects in the Amazon S3 bucket that you used.

```
aws s3api list-objects --bucket bucket
```

The command returns a list of the Amazon S3 objects in bucket that looks like this.


```
{
    "Contents": [
        {
            "Key": "hello-world-job.json",
            "LastModified": "2021-11-18T03:02:12+00:00",
            "ETag": "\"868c8bc3f56b5787964764d4b18ed5ef\"",
            "Size": 54,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
                "ID":
 "e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        },
        {
            "Key": "iot_job_firmware_update.json",
            "LastModified": "2021-04-13T21:57:07+00:00",
            "ETag": "\"7c68c591949391791ecf625253658c61\"",
            "Size": 66,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
                "ID":
 "e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        },
        {
            "Key": "order66.json",
            "LastModified": "2021-04-13T21:57:07+00:00",
            "ETag": "\"bca60d5380b88e1a70cc27d321caba72\"",
            "Size": 29,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "EXAMPLE",
```

```
                "ID":
"e9e3d6ec1EXAMPLEf5bfb5e6bd0a2b6ed03884d1ed392a82ad011c144736a4ee"
            }
        }
    ]
}
```

b. For each object that you recognize from the list as an object you created in this learning path, replace *bucket* with the bucket name and *key* with key value of the object to delete, and then run this command to delete an Amazon S3 object.

```
aws s3api delete-object --bucket bucket --key key
```

If the command is successful, it returns nothing.

After you delete all the AWS resources and objects that you created while completing this learning path, you can start over and repeat the tutorials.

Building solutions with the AWS IoT Device SDKs

The tutorials in this section help walk you through the steps to develop an IoT solution that can be deployed to a production environment using AWS IoT.

These tutorials can take more time to complete than those in the section on the section called "Building demos with the AWS IoT Device Client" because they use the AWS IoT Device SDKs and explain the concepts being applied in more detail to help you create secure and reliable solutions.

Start building solutions with the AWS IoT Device SDKs

These tutorials walk you through different AWS IoT scenarios. Where appropriate, the tutorials use the AWS IoT Device SDKs.

Topics

Tutorial: Connecting a device to AWS IoT Core by using the AWS IoT Device SDK

This tutorial demonstrates how to connect a device to AWS IoT Core so that it can send and receive data to and from AWS IoT. After you complete this tutorial, your device will be configured to connect to AWS IoT Core and you'll understand how devices communicate with AWS IoT.

Topics

Pre-requisites

Before you start this tutorial, make sure that you have:

- **Completed Getting started with AWS IoT Core tutorials**

  In the section of that tutorial where you must the section called "Configure your device", select the the section called "Connect a Raspberry Pi or other device" option for your device and use the Python language options to configure your device.

  > **Note**
  >
  > Keep open the terminal window you use in that tutorial because you'll also use it in this tutorial.

- **A device that can run the AWS IoT Device SDK v2 for Python.**

  This tutorial shows how to connect a device to AWS IoT Core by using Python code examples, which require a relatively powerful device. If you are working with resource-constrained devices, these code examples might not work on them. In that case, you might have more success with the the section called "Using the AWS IoT Device SDK for Embedded C" tutorial.

- **Obtained the required information to connect to the device**

  To connect your device to AWS IoT, you must have information about the thing name, the host name, and the port number.

  > **Note**
  >
  > You can also use custom authentication to connect devices to AWS IoT Core. The connection data you pass to your authorizer Lambda function depends on the protocol you use.

  - **Thing name**: The name of the AWS IoT thing that you want to connect to. You must have registered as your device as an AWS IoT thing. For more information, see Managing devices with AWS IoT.
  - **Host name**: The host name for the account-specific IoT endpoint.
  - **Port number**: The port number to connect to.

  You can use the `configureEndpoint` method in the AWS IoT Python SDK to configure the host name and port number.

```
myAWSIoTMQTTClient.configureEndpoint("random.iot.region.amazonaws.com", 8883)
```

Prepare your device for AWS IoT

In Getting started with AWS IoT Core tutorials, you prepared your device and AWS account so they could communicate. This section reviews the aspects of that preparation that apply to any device connection with AWS IoT Core.

For a device to connect to AWS IoT Core:

1. You must have an **AWS account**.

   The procedure in Set up AWS account describes how to create an AWS account if you don't already have one.

2. In that account, you must have the following **AWS IoT resources** defined for the device in your AWS account and Region.

The procedure in Create AWS IoT resources describes how to create these resources for the device in your AWS account and Region.

- A **device certificate** registered with AWS IoT and activated to authenticate the device.

  The certificate is often created with, and attached to, an **AWS IoT thing object**. While a thing object is not required for a device to connect to AWS IoT, it makes additional AWS IoT features available to the device.
- A **policy** attached to the device certificate that authorizes it to connect to AWS IoT Core and perform all the actions that you want it to.

3. An **internet connection** that can access your AWS account's device endpoints.

   The device endpoints are described in AWS IoT device data and service endpoints and can be seen in the settings page of the AWS IoT console.
4. **Communication software** such as the AWS IoT Device SDKs provide. This tutorial uses the AWS IoT Device SDK v2 for Python.

Review the MQTT protocol

Before we talk about the sample app, it helps to understand the MQTT protocol. The MQTT protocol offers some advantages over other network communication protocols, such as HTTP, which makes it a popular choice for IoT devices. This section reviews the key aspects of MQTT that apply to this tutorial. For information about how MQTT compares to HTTP, see Choosing an application protocol for your device communication.

MQTT uses a publish/subscribe communication model

The MQTT protocol uses a publish/subscribe communication model with its host. This model differs from the request/response model that HTTP uses. With MQTT, devices establish a session with the host that is identified by a unique client ID. To send data, devices publish messages identified by topics to a message broker in the host. To receive messages from the message broker, devices subscribe to topics by sending topic filters in subscription requests to the message broker.

MQTT supports persistent sessions

The message broker receives messages from devices and publishes messages to devices that have subscribed to them. With persistent sessions —sessions that remain active even when the initiating device is disconnected—devices can retrieve messages that were published while they were disconnected. On the device side, MQTT supports Quality of Service levels (QoS) that ensure the host receives messages sent by the device.

Review the pubsub.py Device SDK sample app

This section reviews the `pubsub.py` sample app from the **AWS IoT Device SDK v2 for Python** used in this tutorial. Here, we'll review how it connects to AWS IoT Core to publish and subscribe to MQTT messages. The next section presents some exercises to help you explore how a device connects and communicates with AWS IoT Core.

The pubsub.py sample app demonstrates these aspects of an MQTT connection with AWS IoT Core:

- Communication protocols
- Persistent sessions
- Quality of Service
- Message publish
- Message subscription
- Device disconnection and reconnection

Communication protocols

The `pubsub.py` sample demonstrates an MQTT connection using the MQTT and MQTT over WSS protocols. The AWS common runtime (AWS CRT) library provides the low-level communication protocol support and is included with the AWS IoT Device SDK v2 for Python.

MQTT

The `pubsub.py` sample calls `mtls_from_path` (shown here) in the `mqtt_connection_builder` to establish a connection with AWS IoT Core by using the MQTT protocol. `mtls_from_path` uses X.509 certificates and TLS v1.2 to authenticate the device. The AWS CRT library handles the lower-level details of that connection.

```
mqtt_connection = mqtt_connection_builder.mtls_from_path(
    endpoint=args.endpoint,
    cert_filepath=args.cert,
    pri_key_filepath=args.key,
    ca_filepath=args.ca_file,
    client_bootstrap=client_bootstrap,
```

```
    on_connection_interrupted=on_connection_interrupted,
    on_connection_resumed=on_connection_resumed,
    client_id=args.client_id,
    clean_session=False,
    keep_alive_secs=6
)
```

`endpoint`

Your AWS account's IoT device endpoint

In the sample app, this value is passed in from the command line.

`cert_filepath`

The path to the device's certificate file

In the sample app, this value is passed in from the command line.

`pri_key_filepath`

The path to the device's private key file that was created with its certificate file

In the sample app, this value is passed in from the command line.

`ca_filepath`

The path to the Root CA file. Required only if the MQTT server uses a certificate that's not already in your trust store.

In the sample app, this value is passed in from the command line.

`client_bootstrap`

The common runtime object that handles socket communication activities

In the sample app, this object is instantiated before the call to `mqtt_connection_builder.mtls_from_path`.

`on_connection_interrupted`, `on_connection_resumed`

The callback functions to call when the device's connection is interrupted and resumed

`client_id`

The ID that uniquely identifies this device in the AWS Region

In the sample app, this value is passed in from the command line.

`clean_session`

Whether to start a new persistent session, or, if one is present, reconnect to an existing one

`keep_alive_secs`

The keep alive value, in seconds, to send in the `CONNECT` request. A ping will automatically be sent at this interval. If the server doesn't receive a ping after 1.5 times this value, it assumes that the connection is lost.

MQTT over WSS

The `pubsub.py` sample calls `websockets_with_default_aws_signing` (shown here) in the [`mqtt_connection_builder`]() to establish a connection with AWS IoT Core using the MQTT protocol over WSS. `websockets_with_default_aws_signing` creates an MQTT connection over WSS using [Signature V4]() to authenticate the device.

```
mqtt_connection = mqtt_connection_builder.websockets_with_default_aws_signing(
    endpoint=args.endpoint,
    client_bootstrap=client_bootstrap,
    region=args.signing_region,
    credentials_provider=credentials_provider,
    websocket_proxy_options=proxy_options,
    ca_filepath=args.ca_file,
    on_connection_interrupted=on_connection_interrupted,
    on_connection_resumed=on_connection_resumed,
    client_id=args.client_id,
    clean_session=False,
    keep_alive_secs=6
)
```

`endpoint`

Your AWS account’s IoT device endpoint

In the sample app, this value is passed in from the command line.

`client_bootstrap`

The common runtime object that handles socket communication activities

In the sample app, this object is instantiated before the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.

`region`

The AWS signing Region used by Signature V4 authentication. In `pubsub.py`, it passes the parameter entered in the command line.

In the sample app, this value is passed in from the command line.

`credentials_provider`

The AWS credentials provided to use for authentication

In the sample app, this object is instantiated before the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.

`websocket_proxy_options`

HTTP proxy options, if using a proxy host

In the sample app, this value is initialized before the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.

`ca_filepath`

The path to the Root CA file. Required only if the MQTT server uses a certificate that's not already in your trust store.

In the sample app, this value is passed in from the command line.

`on_connection_interrupted`, `on_connection_resumed`

The callback functions to call when the device’s connection is interrupted and resumed

`client_id`

The ID that uniquely identifies this device in the AWS Region.

In the sample app, this value is passed in from the command line.

`clean_session`

Whether to start a new persistent session, or, if one is present, reconnect to an existing one `keep_alive_secs`

The keep alive value, in seconds, to send in the `CONNECT` request. A ping will automatically be sent at this interval. If the server doesn't receive a ping after 1.5 times this value, it assumes the connection is lost.

HTTPS

What about HTTPS? AWS IoT Core supports devices that publish HTTPS requests. From a programming perspective, devices send HTTPS requests to AWS IoT Core as would any other application. For an example of a Python program that sends an HTTP message from a device, see the HTTPS code example using Python's `requests` library. This example sends a message to AWS IoT Core using HTTPS such that AWS IoT Core interprets it as an MQTT message.

While AWS IoT Core supports HTTPS requests from devices, be sure to review the information about Choosing an application protocol for your device communication so that you can make an informed decision on which protocol to use for your device communications.

Persistent sessions

In the sample app, setting the `clean_session` parameter to `False` indicates that the connection should be persistent. In practice, this means that the connection opened by this call reconnects to an existing persistent session, if one exists. Otherwise, it creates and connects to a new persistent session.

With a persistent session, messages that are sent to the device are stored by the message broker while the device is not connected. When a device reconnects to a persistent session, the message broker sends to the device any stored messages to which it has subscribed.

Without a persistent session, the device will not receive messages that are sent while the device isn't connected. Which option to use depends on your application and whether messages that occur while a device is not connected must be communicated. For more information, see MQTT persistent sessions.

Quality of Service

When the device publishes and subscribes to messages, the preferred Quality of Service (QoS) can be set. AWS IoT supports QoS levels 0 and 1 for publish and subscribe operations. For more information about QoS levels in AWS IoT, see MQTT Quality of Service (QoS) options.

The AWS CRT runtime for Python defines these constants for the QoS levels that it supports:

Python Quality of Service levels

MQTT QoS level	Python symbolic value used by SDK	Description
QoS level 0	`mqtt.QoS.AT_MOST_ONCE`	Only one attempt to send the message will be made, whether it is received or not. The message might not be sent at all, for example, if the device is not connected or there's a network error.
QoS level 1	`mqtt.QoS.AT_LEAST_ONCE`	The message is sent repeatedly until a PUBACK acknowledgement is received.

In the sample app, the publish and subscribe requests are made with a QoS level of 1 (`mqtt.QoS.AT_LEAST_ONCE`).

- **QoS on publish**

  When a device publishes a message with QoS level 1, it sends the message repeatedly until it receives a PUBACK response from the message broker. If the device isn't connected, the message is queued to be sent after it reconnects.

- **QoS on subscribe**

  When a device subscribes to a message with QoS level 1, the message broker saves the messages to which the device is subscribed until they can be sent to the device. The message broker resends the messages until it receives a PUBACK response from the device.

Message publish

After successfully establishing a connection to AWS IoT Core, devices can publish messages. The `pubsub.py` sample does this by calling the `publish` operation of the `mqtt_connection` object.

```
mqtt_connection.publish(
    topic=args.topic,
```

```
    payload=message,
    qos=mqtt.QoS.AT_LEAST_ONCE
)
```

`topic`

The message's topic name that identifies the message

In the sample app, this is passed in from the command line.

`payload`

The message payload formatted as a string (for example, a JSON document)

In the sample app, this is passed in from the command line.

A JSON document is a common payload format, and one that is recognized by other AWS IoT services; however, the data format of the message payload can be anything that the publishers and subscribers agree upon. Other AWS IoT services, however, only recognize JSON, and CBOR, in some cases, for most operations.

`qos`

The QoS level for this message

Message subscription

To receive messages from AWS IoT and other services and devices, devices subscribe to those messages by their topic name. Devices can subscribe to individual messages by specifying a topic name, and to a group of messages by specifying a topic filter, which can include wild card characters. The `pubsub.py` sample uses the code shown here to subscribe to messages and register the callback functions to process the message after it's received.

```
subscribe_future, packet_id = mqtt_connection.subscribe(
    topic=args.topic,
    qos=mqtt.QoS.AT_LEAST_ONCE,
    callback=on_message_received
)
subscribe_result = subscribe_future.result()
```

`topic`

The topic to subscribe to. This can be a topic name or a topic filter.

In the sample app, this is passed in from the command line.

`qos`

Whether the message broker should store these messages while the device is disconnected.

A value of `mqtt.QoS.AT_LEAST_ONCE` (QoS level 1), requires a persistent session to be specified (`clean_session=False`) when the connection is created.

`callback`

The function to call to process the subscribed message.

The `mqtt_connection.subscribe` function returns a future and a packet ID. If the subscription request was initiated successfully, the packet ID returned is greater than 0. To make sure that the subscription was received and registered by the message broker, you must wait for the result of the asynchronous operation to return, as shown in the code example.

The callback function

The callback in the `pubsub.py` sample processes the subscribed messages as the device receives them.

```
def on_message_received(topic, payload, **kwargs):
    print("Received message from topic '{}': {}".format(topic, payload))
    global received_count
    received_count += 1
    if received_count == args.count:
        received_all_event.set()
```

`topic`

The message's topic

This is the specific topic name of the message received, even if you subscribed to a topic filter.

`payload`

The message payload

The format for this is application specific.

`kwargs`

Possible additional arguments as described in [`mqtt.Connection.subscribe`](#).

In the `pubsub.py` sample, `on_message_received` only displays the topic and its payload. It also counts the messages received to end the program after the limit is reached.

Your app would evaluate the topic and the payload to determine what actions to perform.

Device disconnection and reconnection

The `pubsub.py` sample includes callback functions that are called when the device is disconnected and when the connection is re-established. What actions your device takes on these events is application specific.

When a device connects for the first time, it must subscribe to topics to receive. If a device's session is present when it reconnects, its subscriptions are restored, and any stored messages from those subscriptions are sent to the device after it reconnects.

If a device's session no longer exists when it reconnects, it must resubscribe to its subscriptions. Persistent sessions have a limited lifetime and can expire when the device is disconnected for too long.

Connect your device and communicate with AWS IoT Core

This section presents some exercises to help you explore different aspects of connecting your device to AWS IoT Core. For these exercises, you'll use the MQTT test client in the AWS IoT console to see what your device publishes and to publish messages to your device. These exercises use the `pubsub.py` sample from the AWS IoT Device SDK v2 for Python and build on your experience with *Get started tutorials* tutorials.

In this section, you'll:

- Subscribe to wild card topic filters
- Process topic filter subscriptions
- Publish messages from your device

For these exercises, you'll start from the `pubsub.py` sample program.

> **Note**
>
> These exercises assume that you completed the *Get started tutorials* tutorials and use the terminal window for your device from that tutorial.

Subscribe to wild card topic filters

In this exercise, you'll modify the command line used to call `pubsub.py` to subscribe to a wild card topic filter and process the messages received based on the message's topic.

Exercise procedure

For this exercise, imagine that your device contains a temperature control and a light control. It uses these topic names to identify the messages about them.

1. Before starting the exercise, try running this command from the *Get started tutorials* tutorials on your device to make sure that everything is ready for the exercise.

```
cd ~/aws-iot-device-sdk-python-v2/samples
python3 pubsub.py --topic topic_1 --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/
certs/device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint
```

   You should see the same output as you saw in the Getting started tutorial.

2. For this exercise, change these command line parameters.

Action	Command line parameter	Effect
add	**--message** `""`	Configure `pubsub.py` to listen only
add	**--count** `2`	End the program after receiving two messages
change	**--topic** `device/+/details`	Define the topic filter to subscribe to

   Making these changes to the initial command line results in this command line. Enter this command in the terminal window for your device.

```
python3 pubsub.py --message "" --count 2 --topic device/+/details --ca_file
 ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/
private.pem.key --endpoint your-iot-endpoint
```

The program should display something like this:

```
Connecting to a3qexamplesffp-ats.iot.us-west-2.amazonaws.com with client ID
 'test-24d7cdcc-cc01-458c-8488-2d05849691e1'...
Connected!
Subscribing to topic 'device/+/details'...
Subscribed with QoS.AT_LEAST_ONCE
Waiting for all messages to be received...
```

If you see something like this on your terminal, your device is ready and listening for messages where the topic names start with `device` and end with `/detail`. So, let's test that.

3. Here are a couple of messages that your device might receive.

Topic name	Message payload
`device/temp/details`	`{ "desiredTemp": 20, "currentTemp": 15 }`
`device/light/details`	`{ "desiredLight": 100, "currentLight": 50 }`

4. Using the MQTT test client in the AWS IoT console, send the messages described in the previous step to your device.

   a. Open the MQTT test client in the AWS IoT console.

   b. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: **`device/+/details`**, and then choose **Subscribe to topic**.

   c. In the **Subscriptions** column of the MQTT test client, choose **device/+/details**.

   d. For each of the topics in the preceding table, do the following in the MQTT test client:

      1. In **Publish**, enter the value from the **Topic name** column in the table.
      2. In the message payload field below the topic name, enter the value from the **Message payload** column in the table.
      3. Watch the terminal window where `pubsub.py` is running and, in the MQTT test client, choose **Publish to topic**.

You should see that the message was received by pubsub.py in the terminal window.

Exercise result

With this, pubsub.py, subscribed to the messages using a wild card topic filter, received them, and displayed them in the terminal window. Notice how you subscribed to a single topic filter, and the callback function was called to process messages having two distinct topics.

Process topic filter subscriptions

Building on the previous exercise, modify the pubsub.py sample app to evaluate the message topics and process the subscribed messages based on the topic.

Exercise procedure

To evaluate the message topic

1. Copy pubsub.py to pubsub2.py.
2. Open pubsub2.py in your favorite text editor or IDE.
3. In pubsub2.py, find the on_message_received function.
4. In on_message_received, insert the following code after the line that starts with print("Received message and before the line that starts with global received_count.

```
    topic_parsed = False
    if "/" in topic:
        parsed_topic = topic.split("/")
        if len(parsed_topic) == 3:
            # this topic has the correct format
            if (parsed_topic[0] == 'device') and (parsed_topic[2] == 'details'):
                # this is a topic we care about, so check the 2nd element
                if (parsed_topic[1] == 'temp'):
                    print("Received temperature request: {}".format(payload))
                    topic_parsed = True
                if (parsed_topic[1] == 'light'):
                    print("Received light request: {}".format(payload))
                    topic_parsed = True
    if not topic_parsed:
```

```
        print("Unrecognized message topic.")
```

5. Save your changes and run the modified program by using this command line.

```
python3 pubsub2.py --message "" --count 2 --topic device/+/details --ca_file
 ~/certs/Amazon-root-CA-1.pem --cert ~/certs/device.pem.crt --key ~/certs/
private.pem.key --endpoint your-iot-endpoint
```

6. In the AWS IoT console, open the MQTT test client.
7. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: **`device/+/details`**, and then choose **Subscribe to topic**.
8. In the **Subscriptions** column of the MQTT test client, choose **device/+/details**.
9. For each of the topics in this table, do the following in the MQTT test client:

Topic name	Message payload
`device/temp/details`	`{ "desiredTemp": 20, "currentTemp": 15 }`
`device/light/details`	`{ "desiredLight": 100, "currentLight": 50 }`

   1. In **Publish**, enter the value from the **Topic name** column in the table.
   2. In the message payload field below the topic name, enter the value from the **Message payload** column in the table.
   3. Watch the terminal window where `pubsub.py` is running and, in the MQTT test client, choose **Publish to topic**.

   You should see that the message was received by `pubsub.py` in the terminal window.

You should see something similar to this in your terminal window.

```
Connecting to a3qexamplesffp-ats.iot.us-west-2.amazonaws.com with client ID 'test-
af794be0-7542-45a0-b0af-0b0ea7474517'...
Connected!
Subscribing to topic 'device/+/details'...
```

```
Subscribed with QoS.AT_LEAST_ONCE
Waiting for all messages to be received...
Received message from topic 'device/light/details': b'{ "desiredLight": 100,
 "currentLight": 50 }'
Received light request: b'{ "desiredLight": 100, "currentLight": 50 }'
Received message from topic 'device/temp/details': b'{ "desiredTemp": 20,
 "currentTemp": 15 }'
Received temperature request: b'{ "desiredTemp": 20, "currentTemp": 15 }'
2 message(s) received.
Disconnecting...
Disconnected!
```

Exercise result

In this exercise, you added code so the sample app would recognize and process multiple messages in the callback function. With this, your device could receive messages and act on them.

Another way for your device to receive and process multiple messages is to subscribe to different messages separately and assign each subscription to its own callback function.

Publish messages from your device

You can use the pubsub.py sample app to publish messages from your device. While it will publish messages as it is, the messages can't be read as JSON documents. This exercise modifies the sample app to be able to publish JSON documents in the message payload that can be read by AWS IoT Core.

Exercise procedure

In this exercise, the following message will be sent with the `device/data` topic.

```
{
    "timestamp": 1601048303,
    "sensorId": 28,
    "sensorData": [
        {
        "sensorName": "Wind speed",
        "sensorValue": 34.2211224
        }
    ]
}
```

To prepare your MQTT test client to monitor the messages from this exercise

1. In **Subscribe to a topic**, in the **Subscription topic field**, enter the topic filter: **`device/data`**, and then choose **Subscribe to topic**.
2. In the **Subscriptions** column of the MQTT test client, choose **device/data**.
3. Keep the MQTT test client window open to wait for messages from your device.

To send JSON documents with the pubsub.py sample app

1. On your device, copy `pubsub.py` to `pubsub3.py`.
2. Edit `pubsub3.py` to change how it formats the messages it publishes.
   a. Open `pubsub3.py` in a text editor.
   b. Locate this line of code:

      ```
      message = "{} [{}]".format(message_string, publish_count)
      ```
   c. Change it to:

      ```
      message = "{}".format(message_string)
      ```
   d. Locate this line of code:

      ```
      message_json = json.dumps(message)
      ```
   e. Change it to:

      ```
      message = "{}".json.dumps(json.loads(message))
      ```
   f. Save your changes.
3. On your device, run this command to send the message two times.

   ```
   python3 pubsub3.py  --ca_file ~/certs/Amazon-root-CA-1.pem  --cert ~/certs/device.pem.crt  --key ~/certs/private.pem.key  --topic device/data  --count 2 --message '{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind speed","sensorValue":34.2211224}]}'  --endpoint your-iot-endpoint
   ```
4. In the MQTT test client, check to see that it has interpreted and formatted the JSON document in the message payload, such as this:

By default, pubsub3.py also subscribes to the messages it sends. You should see that it received the messages in the app's output. The terminal window should look something like this.

```
Connecting to a3qEXAMPLEsffp-ats.iot.us-west-2.amazonaws.com with client ID
 'test-5cff18ae-1e92-4c38-a9d4-7b9771afc52f'...
Connected!
Subscribing to topic 'device/data'...
Subscribed with QoS.AT_LEAST_ONCE
Sending 2 message(s)
Publishing message to topic 'device/data':
 {"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind
 speed","sensorValue":34.2211224}]}
Received message from topic 'device/data':
 b'{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind
 speed","sensorValue":34.2211224}]}'
Publishing message to topic 'device/data':
 {"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind
 speed","sensorValue":34.2211224}]}
Received message from topic 'device/data':
 b'{"timestamp":1601048303,"sensorId":28,"sensorData":[{"sensorName":"Wind
 speed","sensorValue":34.2211224}]}'
2 message(s) received.
Disconnecting...
Disconnected!
```

Exercise result

With this, your device can generate messages to send to AWS IoT Core to test basic connectivity and provide device messages for AWS IoT Core to process. For example, you could use this app to send test data from your device to test AWS IoT rule actions.

Review the results

The examples in this tutorial gave you hands-on experience with the basics of how devices can communicate with AWS IoT Core—a fundamental part of your AWS IoT solution. When your devices are able to communicate with AWS IoT Core, they can pass messages to AWS services and other devices on which they can act. Likewise, AWS services and other devices can process information that results in messages sent back to your devices.

When you are ready to explore AWS IoT Core further, try these tutorials:

- the section called “Sending an Amazon SNS notification”
- the section called “Storing device data in a DynamoDB table”
- the section called “Formatting a notification by using an AWS Lambda function”

Tutorial: Using the AWS IoT Device SDK for Embedded C

This section describes how to run the AWS IoT Device SDK for Embedded C.

Procedures in this section

- Step1: Install the AWS IoT Device SDK for Embedded C
- Step 2: Configure the sample app
- Step 3: Build and run the sample application

Step1: Install the AWS IoT Device SDK for Embedded C

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs). If you have more memory and processing resources available, we recommend that you use one of the higher order AWS IoT Device and Mobile SDKs (for example, C++, Java, JavaScript, and Python).

In general, the AWS IoT Device SDK for Embedded C is intended for systems that use MCUs or low-end MPUs that run embedded operating systems. For the programming example in this section, we assume your device uses Linux.

Example

1. Download the AWS IoT Device SDK for Embedded C to your device from GitHub.

```
git clone https://github.com/aws/aws-iot-device-sdk-embedded-c.git --recurse-submodules
```

   This creates a directory named `aws-iot-device-sdk-embedded-c` in the current directory.

2. Navigate to that directory and checkout the latest release. Please see github.com/aws/aws-iot-device-sdk-embedded-C/tags for the latest release tag.

```
cd aws-iot-device-sdk-embedded-c
git checkout latest-release-tag
```

3. Install OpenSSL version 1.1.0 or later. The OpenSSL development libraries are usually called "libssl-dev" or "openssl-devel" when installed through a package manager.

```
sudo apt-get install libssl-dev
```

Step 2: Configure the sample app

The AWS IoT Device SDK for Embedded C includes sample applications for you to try. For simplicity, this tutorial uses the `mqtt_demo_mutual_auth` application, that illustrates how to connect to the AWS IoT Core message broker and subscribe and publish to MQTT topics.

1. Copy the certificate and private key you created in Getting started with AWS IoT Core tutorials into the `build/bin/certificates` directory.

> **Note**
>
> Device and root CA certificates are subject to expiration or revocation. If these certificates expire or are revoked, you must copy a new CA certificate or private key and device certificate onto your device.

2. You must configure the sample with your personal AWS IoT Core endpoint, private key, certificate, and root CA certificate. Navigate to the `aws-iot-device-sdk-embedded-c/demos/mqtt/mqtt_demo_mutual_auth` directory.

   If you have the AWS CLI installed, you can use this command to find your account's endpoint URL.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   If you don't have the AWS CLI installed, open your [AWS IoT console](). From the navigation pane, choose **Manage**, and then choose **Things**. Choose the IoT thing for your device, and then choose **Interact**. Your endpoint is displayed in the **HTTPS** section of the thing details page.

3. Open the `demo_config.h` file and update the values for the following:

   AWS_IOT_ENDPOINT

      Your personal endpoint.

   CLIENT_CERT_PATH

      Your certificate file path, for example `certificates/device.pem.crt"`.

   CLIENT_PRIVATE_KEY_PATH

      Your private key file name, for example `certificates/private.pem.key`.

   For example:

   ```
   // Get from demo_config.h
   // =================================================
   #define AWS_IOT_ENDPOINT               "my-endpoint-ats.iot.us-
   east-1.amazonaws.com"
   #define AWS_MQTT_PORT                  8883
   #define CLIENT_IDENTIFIER              "testclient"
   #define ROOT_CA_CERT_PATH              "certificates/AmazonRootCA1.crt"
   #define CLIENT_CERT_PATH               "certificates/my-device-cert.pem.crt"
   #define CLIENT_PRIVATE_KEY_PATH        "certificates/my-device-private-key.pem.key"
   // =================================================
   ```

4. Check to see if you have CMake installed on your device by using this command.

```
cmake --version
```

If you see the version information for the compiler, you can continue to the next section.

If you get an error or don't see any information, then you'll need to install the cmake package using this command.

```
sudo apt-get install cmake
```

Run the **cmake --version** command again and confirm that CMake has been installed and that you are ready to continue.

5. Check to see if you have the development tools installed on your device by using this command.

```
gcc --version
```

If you see the version information for the compiler, you can continue to the next section.

If you get an error or don't see any compiler information, you'll need to install the `build-essential` package using this command.

```
sudo apt-get install build-essential
```

Run the **gcc --version** command again and confirm that the build tools have been installed and that you are ready to continue.

Step 3: Build and run the sample application

This procedure exaplains how to generate the `mqtt_demo_mutual_auth` application on your device and connect it to the [AWS IoT console](https://console.aws.amazon.com/iot/home) using the AWS IoT Device SDK for Embedded C.

To run the AWS IoT Device SDK for Embedded C sample applications

1. Navigate to `aws-iot-device-sdk-embedded-c` and create a build directory.

```
mkdir build && cd build
```

2. Enter the following CMake command to generate the Makefiles needed to build.

```
cmake ..
```

3. Enter the following command to build the executable app file.

```
make
```

4. Run the `mqtt_demo_mutual_auth` app with this command.

```
cd bin
./mqtt_demo_mutual_auth
```

You should see output similar to the following:

```
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:584] Establishing a TLS session to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com:8883.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1264] Creating an MQTT connection to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com.
[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=2.
[INFO] [MQTT] [core_mqtt_serializer.c:970] CONNACK session present bit not set.
[INFO] [MQTT] [core_mqtt_serializer.c:912] Connection accepted.
[INFO] [MQTT] [core_mqtt.c:1526] Received MQTT CONNACK successfully from broker.
[INFO] [MQTT] [core_mqtt.c:1792] MQTT connection established with the broker.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1033] MQTT connection successfully established with broker.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1296] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1314] Subscribing to the MQTT topic testclient/example/topic.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1097] SUBSCRIBE sent for topic testclient/example/topic to broker.


[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=3.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:921] Subscribed to the topic testclient/example/topic. with maximum QoS 1.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1358] Sending Publish to the MQTT topic testclient/example/topic.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1195] PUBLISH sent for topic testclient/example/topic to broker with packet ID 2.


[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=2.
[INFO] [MQTT] [core_mqtt.c:1126] Ack packet deserialized with result: MQTTSuccess.
[INFO] [MQTT] [core_mqtt.c:1139] State record updated. New state=MQTTPublishDone.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:946] PUBACK received for packet id 2.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:672] Cleaned up outgoing publish packet with packet id 2.


[INFO] [MQTT] [core_mqtt.c:855] Packet received. ReceivedBytes=40.
[INFO] [MQTT] [core_mqtt.c:1015] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
```

Your device is now connected to AWS IoT using the AWS IoT Device SDK for Embedded C.

You can also use the AWS IoT console to view the MQTT messages that the sample app is publishing. For information about how to use the MQTT client in the AWS IoT console, see the section called "View MQTT messages with the AWS IoT MQTT client" .

Creating AWS IoT rules to route device data to other services

These tutorials show you how to create and test AWS IoT rules using some of the more common rule actions.

AWS IoT rules send data from your devices to other AWS services. They listen for specific MQTT messages, format the data in the message payloads, and send the result to other AWS services.

We recommend that you try these in the order they are shown here, even if your goal is to create a rule that uses a Lambda function or something more complex. The tutorials are presented in order from basic to complex. They present new concepts incrementally to help you learn the concepts you can use to create the rule actions that don't have a specific tutorial.

> **Note**
>
> AWS IoT rules help you send the data from your IoT devices to other AWS services. To do that successfully, however, you need a working knowledge of the other services where you want to send data. While these tutorials provide the necessary information to complete the tasks, you might find it helpful to learn more about the services you want to send data to before you use them in your solution. A detailed explanation of the other AWS services is outside of the scope of these tutorials.

Tutorial scenario overview

The scenario for these tutorials is that of a weather sensor device that periodically publishes its data. There are many such sensor devices in this imaginary system. The tutorials in this section, however, focus on a single device while showing how you might accommodate multiple sensors.

The tutorials in this section show you how to use AWS IoT rules to do the following tasks with this imaginary system of weather sensor devices.

- **Tutorial: Republishing an MQTT message**

  This tutorial shows how to republish an MQTT message received from the weather sensors as a message that contains only the sensor ID and the temperature value. It uses only AWS IoT Core services and demonstrates a simple SQL query and how to use the MQTT client to test your rule.
- **Tutorial: Sending an Amazon SNS notification**

This tutorial shows how to send an SNS message when a value from a weather sensor device exceeds a specific value. It builds on the concepts presented in the previous tutorial and adds how to work with another AWS service, the Amazon Simple Notification Service (Amazon SNS).

If you're new to Amazon SNS, review its Getting started exercises before you start this tutorial.

- **Tutorial: Storing device data in a DynamoDB table**

  This tutorial shows how to store the data from the weather sensor devices in a database table. It uses the rule query statement and substitution templates to format the message data for the destination service, Amazon DynamoDB.

  If you're new to DynamoDB, review its Getting started exercises before you start this tutorial.

- **Tutorial: Formatting a notification by using an AWS Lambda function**

  This tutorial shows how to call a Lambda function to reformat the device data and then send it as a text message. It adds a Python script and AWS SDK functions in an AWS Lambda function to format with the message payload data from the weather sensor devices and send a text message.

  If you're new to Lambda, review its Getting started exercises before you start this tutorial.

AWS IoT rule overview

All of these tutorials create AWS IoT rules.

For an AWS IoT rule to send the data from a device to another AWS service, it uses:

- A rule query statement that consists of:
  - A SQL SELECT clause that selects and formats the data from the message payload
  - A topic filter (the FROM object in the rule query statement) that identifies the messages to use
  - An optional conditional statement (a SQL WHERE clause) that specifies specific conditions on which to act
- At least one rule action

Devices publish messages to MQTT topics. The topic filter in the SQL SELECT statement identifies the MQTT topics to apply the rule to. The fields specified in the SQL SELECT statement format the data from the incoming MQTT message payload for use by the rule's actions. For a complete list of rule actions, see AWS IoT Rule Actions.

Tutorials in this section

- Tutorial: Republishing an MQTT message
- Tutorial: Sending an Amazon SNS notification
- Tutorial: Storing device data in a DynamoDB table
- Tutorial: Formatting a notification by using an AWS Lambda function

Tutorial: Republishing an MQTT message

This tutorial demonstrates how to create an AWS IoT rule that publishes an MQTT message when a specified MQTT message is received. The incoming message payload can be modified by the rule before it's published. This makes it possible to create messages that are tailored to specific applications without the need to alter your device or its firmware. You can also use the filtering aspect of a rule to publish messages only when a specific condition is met.

The messages republished by a rule act like messages sent by any other AWS IoT device or client. Devices can subscribe to the republished messages the same way they can subscribe to any other MQTT message topic.

What you'll learn in this tutorial:

- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- Review MQTT topics and AWS IoT rules
- Step 1: Create an AWS IoT rule to republish an MQTT message
- Step 2: Test your new rule
- Step 3: Review the results and next steps

Before you start this tutorial, make sure that you have:

- **Set up AWS account**

  You'll need your AWS account and AWS IoT console to complete this tutorial.

- **Reviewed View MQTT messages with the AWS IoT MQTT client**

  Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

Review MQTT topics and AWS IoT rules

Before talking about AWS IoT rules, it helps to understand the MQTT protocol. In IoT solutions, the MQTT protocol offers some advantages over other network communication protocols, such as HTTP, which makes it a popular choice for use by IoT devices. This section reviews the key aspects of MQTT as they apply to this tutorial. For information about how MQTT compares to HTTP, see Choosing an application protocol for your device communication.

MQTT protocol

The MQTT protocol uses a publish/subscribe communication model with its host. To send data, devices publish messages that are identified by topics to the AWS IoT message broker. To receive messages from the message broker, devices subscribe to the topics they will receive by sending topic filters in subscription requests to the message broker. The AWS IoT rules engine receives MQTT messages from the message broker.

AWS IoT rules

AWS IoT rules consist of a rule query statement and one or more rule actions. When the AWS IoT rules engine receives an MQTT message, these elements act on the message as follows.

- **Rule query statement**

  The rule's query statement describes the MQTT topics to use, interprets the data from the message payload, and formats the data as described by a SQL statement that is similar to statements used by popular SQL databases. The result of the query statement is the data that is sent to the rule's actions.

- **Rule action**

Each rule action in a rule acts on the data that results from the rule's query statement. AWS IoT supports many rule actions. In this tutorial, however, you'll concentrate on the Republish rule action, which publishes the result of the query statement as an MQTT message with a specific topic.

Step 1: Create an AWS IoT rule to republish an MQTT message

The AWS IoT rule that you'll create in this tutorial subscribes to the `device/device_id/data` MQTT topics where *device_id* is the ID of the device that sent the message. These topics are described by a topic filter as `device/+/data`, where the + is a wildcard character that matches any string between the two forward slash characters.

When the rule receives a message from a matching topic, it republishes the `device_id` and `temperature` values as a new MQTT message with the `device/data/temp` topic.

For example, the payload of an MQTT message with the `device/22/data` topic looks like this:

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

The rule takes the `temperature` value from the message payload, and the `device_id` from the topic, and republishes them as an MQTT message with the `device/data/temp` topic and a message payload that looks like this:

```
{
  "device_id": "22",
  "temperature": 28
}
```

With this rule, devices that only need the device's ID and the temperature data subscribe to the `device/data/temp` topic to receive only that information.

To create a rule that republishes an MQTT message

1. Open [the **Rules** hub of the AWS IoT console]().
2. In **Rules**, choose **Create** and start creating your new rule.
3. In the top part of **Create a rule**:

   a. In **Name**, enter the rule's name. For this tutorial, name it **`republish_temp`**.

      Remember that a rule name must be unique within your Account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.

   b. In **Description**, describe the rule.

      A meaningful description helps you remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.
4. In **Rule query statement** of **Create a rule**:

   a. In **Using SQL version**, select **`2016-03-23`**.

   b. In the **Rule query statement** edit box, enter the statement:

      ```
      SELECT topic(2) as device_id, temperature FROM 'device/+/data'
      ```

      This statement:

      - Listens for MQTT messages with a topic that matches the `device/+/data` topic filter.
      - Selects the second element from the topic string and assigns it to the `device_id` field.
      - Selects the value `temperature` field from the message payload and assigns it to the `temperature` field.
5. In **Set one or more actions**:

   a. To open up the list of rule actions for this rule, choose **Add action**.

   b. In **Select an action**, choose **Republish a message to an AWS IoT topic**.

   c. At the bottom of the action list, choose **Configure action** to open the selected action's configuration page.
6. In **Configure action**:

a. In **Topic**, enter **`device/data/temp`**. This is the MQTT topic of the message that this rule will publish.

b. In **Quality of Service**, choose **0 - The message is delivered zero or more times**.

c. In **Choose or create a role to grant AWS IoT access to perform this action**:

   i. Choose **Create Role**. The **Create a new role** dialog box opens.

   ii. Enter a name that describes the new role. In this tutorial, use **`republish_role`**.

      When you create a new role, the correct policies to perform the rule action are created and attached to the new role. If you change the topic of this rule action or use this role in another rule action, you must update the policy for that role to authorize the new topic or action. To update an existing role, choose **Update role** in this section.

   iii. Choose **Create Role** to create the role and close the dialog box.

d. Choose **Add action** to add the action to the rule and return to the **Create a rule** page.

7. The **Republish a message to an AWS IoT topic** action is now listed in **Set one or more actions**.

   In the new action's tile, below **Republish a message to an AWS IoT topic**, you can see the topic to which your republish action will publish.

   This is the only rule action you'll add to this rule.

8. In **Create a rule**, scroll down to the bottom and choose **Create rule** to create the rule and complete this step.

Step 2: Test your new rule

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the [MQTT client in the AWS IoT console](#) in a new window. This will let you edit the rule without losing the configuration of your MQTT client. The MQTT client does not retain any subscriptions or message logs if you leave it to go to another page in the console.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topics, in this case, `device/+/data`.

   a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.
   b. In **Subscription topic**, enter the topic of the input topic filter, **`device/+/data`**.
   c. Keep the rest of the fields at their default settings.
   d. Choose **Subscribe to topic**.

      In the **Subscriptions** column, under **Publish to a topic**, **`device/+/data`** appears.

2. Subscribe to the topic that your rule will publish: `device/data/temp`.

   a. Under **Subscriptions**, choose **Subscribe to a topic** again, and in **Subscription topic**, enter the topic of the republished message, **`device/data/temp`**.
   b. Keep the rest of the fields at their default settings.
   c. Choose **Subscribe to topic**.

      In the **Subscriptions** column, under **device/+/data**, **`device/data/temp`** appears.

3. Publish a message to the input topic with a specific device ID, **`device/22/data`**. You can't publish to MQTT topics that contain wildcard characters.

   a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.
   b. In the **Publish** field, enter the input topic name, **`device/22/data`**.
   c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

   d. To send your MQTT message, choose **Publish to topic**.

4. Review the messages that were sent.

   a. In the MQTT client, under **Subscriptions**, there is a green dot next to the two topics to which you subscribed earlier.

The green dots indicate that one or more new messages have been received since the last time you looked at them.

b. Under **Subscriptions**, choose **device/+/data** to check that the message payload matches what you just published and looks like this:

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

c. Under **Subscriptions**, choose **device/data/temp** to check that your republished message payload looks like this:

```
{
  "device_id": "22",
  "temperature": 28
}
```

Notice that the `device_id` value is a quoted string and the `temperature` value is numeric. This is because the [`topic()`](#) function extracted the string from the input message's topic name while the `temperature` value uses the numeric value from the input message's payload.

If you want to make the `device_id` value a numeric value, replace `topic(2)` in the rule query statement with:

```
cast(topic(2) AS DECIMAL)
```

Note that casting the `topic(2)` value to a numeric value will only work if that part of the topic contains only numeric characters.

5. If you see that the correct message was published to the **device/data/temp** topic, then your rule worked. See what more you can learn about the Republish rule action in the next section.

If you don't see that the correct message was published to either the **device/+/data** or **device/data/temp** topics, check the troubleshooting tips.

Troubleshooting your Republish message rule

Here are some things to check in case you're not seeing the results you expect.

- **You got an error banner**

  If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

  Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client if you subscribed to the `device/+/data` topic filter as described in the procedure.

  Things to check

  - **Check the topic filter you subscribed to**

    If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

    If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

  - **Check the message publish function**

    In the MQTT client, under **Subscriptions**, choose **device/+/data**, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't see your republished message in the MQTT client**

  For your rule to work, it must have the correct policy that authorizes it to receive and republish a message and it must receive the message.

  Things to check

  - **Check the AWS Region of your MQTT client and the rule that you created**

The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

- **Check the input message topic in the rule query statement**

  For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

  Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

- **Check the contents of the input message payload**

  For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

  Check the spelling of the `temperature` field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the `temperature` field in the rule query statement must be identical to the `temperature` field in the message payload.

  Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the republished message topic in the rule action**

  The topic to which the Republish rule action publishes the new message must match the topic to which you subscribed in the MQTT client.

  Open the rule you created in the console and check the topic to which the rule action will republish the message.

- **Check the role being used by the rule**

  The rule action must have permission to receive the original topic and publish the new topic.

  The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 3: Review the results and next steps

In this tutorial

- You used a simple SQL query and a couple of functions in a rule query statement to produce a new MQTT message.
- You created a rule that republished that new message.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you republish a few messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the republished message. Here are some ideas to get you started.

- Change the *`device_id`* in the input message's topic and observe the effect in the republished message payload.
- Change the fields selected in the rule query statement and observe the effect in the republished message payload.
- Try the next tutorial in this series and learn how to Tutorial: Sending an Amazon SNS notification.

The Republish rule action used in this tutorial can also help you debug rule query statements. For example, you can add this action to a rule to see how its rule query statement is formatting the data used by its rule actions.

Tutorial: Sending an Amazon SNS notification

This tutorial demonstrates how to create an AWS IoT rule that sends MQTT message data to an Amazon SNS topic so that it can be sent as an SMS text message.

In this tutorial, you create a rule that sends message data from a weather sensor to all subscribers of an Amazon SNS topic, whenever the temperature exceeds the value set in the rule. The rule detects when the reported temperature exceeds the value set by the rule, and then creates a new message payload that includes only the device ID, the reported temperature, and the temperature limit that was exceeded. The rule sends the new message payload as a JSON document to an SNS topic, which notifies all subscribers to the SNS topic.

What you'll learn in this tutorial:

- How to create and test an Amazon SNS notification
- How to call an Amazon SNS notification from an AWS IoT rule
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- Step 1: Create an Amazon SNS topic that sends a SMS text message
- Step 2: Create an AWS IoT rule to send the text message
- Step 3: Test the AWS IoT rule and Amazon SNS notification
- Step 4: Review the results and next steps

Before you start this tutorial, make sure that you have:

- **Set up AWS account**

  You'll need your AWS account and AWS IoT console to complete this tutorial.
- **Reviewed View MQTT messages with the AWS IoT MQTT client**

  Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.
- **Reviewed the Amazon Simple Notification Service**

  If you haven't used Amazon SNS before, review Setting up access for Amazon SNS. If you've already completed other AWS IoT tutorials, your AWS account should already be configured correctly.

Step 1: Create an Amazon SNS topic that sends a SMS text message

This procedure exaplains how to create the Amazon SNS topic your weather sensor can send message data to. The Amazon SNS topic will then notify all of its subscribers via a SMS text message of the temperature limit that was exceeded.

To create an Amazon SNS topic that sends an SMS text message

1. **Create an Amazon SNS topic.**

   a. Sign in to the [Amazon SNS console]().

   b. In the left navigation pane, choose **Topics**.

   c. On the **Topics** page, choose **Create topic**.

   d. In **Details**, choose the **Standard** type. By default, the console creates a FIFO topic.

   e. In **Name**, enter the SNS topic name. For this tutorial, enter **`high_temp_notice`**.

   f. Scroll to the end of the page and choose **Create topic**.

      The console opens the new topic's **Details** page.

2. **Create an Amazon SNS subscription.**

   > **Note**
   >
   > The phone number that you use in this subscription might incur text messaging charges from the messages you will send in this tutorial.

   a. In the **high_temp_notice** topic's details page, choose **Create subscription**.

   b. In **Create subscription**, in the **Details** section, in the **Protocol** list, choose **SMS**.

   c. In **Endpoint**, enter the number of a phone that can receive text messages. Be sure to enter it such that it starts with a +, includes the country and area code, and doesn't include any other punctuation characters.

   d. Choose **Create subscription**.

3. **Test the Amazon SNS notification.**

   a. In the [Amazon SNS console](), in the left navigation pane, choose **Topics**.

   b. To open the topic's details page, in **Topics**, in the list of topics, choose **high_temp_notice**.

c. To open the **Publish message to topic** page, in the **high_temp_notice** details page, choose **Publish message**.

d. In **Publish message to topic**, in the **Message body** section, in **Message body to send to the endpoint**, enter a short message.

e. Scroll down to the bottom of the page and choose **Publish message**.

f. On the phone with the number you used earlier when creating the subscription, confirm that the message was received.

If you did not receive the test message, double check the phone number and your phone's settings.

Make sure you can publish test messages from the Amazon SNS console before you continue the tutorial.

Step 2: Create an AWS IoT rule to send the text message

The AWS IoT rule that you'll create in this tutorial subscribes to the `device/*device_id*/data` MQTT topics where *device_id* is the ID of the device that sent the message. These topics are described in a topic filter as `device/+/data`, where the `+` is a wildcard character that matches any string between the two forward slash characters. This rule also tests the value of the `temperature` field in the message payload.

When the rule receives a message from a matching topic, it takes the *device_id* from the topic name, the `temperature` value from the message payload, and adds a constant value for the limit it's testing, and sends these values as a JSON document to an Amazon SNS notification topic.

For example, an MQTT message from weather sensor device number 32 uses the `device/32/data` topic and has a message payload that looks like this:

```
{
  "temperature": 38,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

The rule's rule query statement takes the `temperature` value from the message payload, the *`device_id`* from the topic name, and adds the constant `max_temperature` value to send a message payload that looks like this to the Amazon SNS topic:

```
{
  "device_id": "32",
  "reported_temperature": 38,
  "max_temperature": 30
}
```

To create an AWS IoT rule to detect an over-limit temperature value and create the data to send to the Amazon SNS topic

1. Open [the **Rules** hub of the AWS IoT console](https://console.aws.amazon.com/iot/home#/rulehub).
2. If this is your first rule, choose **Create**, or **Create a rule**.
3. In **Create a rule**:
   a. In **Name**, enter **`temp_limit_notify`**.

      Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the words in the rule's name.
   b. In **Description**, describe the rule.

      A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.
4. In **Rule query statement** of **Create a rule**:
   a. In **Using SQL version**, select **2016-03-23**.
   b. In the **Rule query statement** edit box, enter the statement:

      ```
      SELECT topic(2) as device_id,
          temperature as reported_temperature,
          30 as max_temperature
        FROM 'device/+/data'
        WHERE temperature > 30
      ```

      This statement:

- Listens for MQTT messages with a topic that matches the `device/+/data` topic filter and that have a `temperature` value greater than 30.
- Selects the second element from the topic string and assigns it to the `device_id` field.
- Selects the value `temperature` field from the message payload and assigns it to the `reported_temperature` field.
- Creates a constant value `30` to represent the limit value and assigns it to the `max_temperature` field.

5. To open up the list of rule actions for this rule, in **Set one or more actions**, choose **Add action**.
6. In **Select an action**, choose **Send a message as an SNS push notification**.
7. To open the selected action's configuration page, at the bottom of the action list, choose **Configure action**.
8. In **Configure action**:

   a. In **SNS target**, choose **Select**, find your SNS topic named **high_temp_notice**, and choose **Select**.

   b. In **Message format**, choose **RAW**.

   c. In **Choose or create a role to grant AWS IoT access to perform this action**, choose **Create Role**.

   d. In **Create a new role**, in **Name**, enter a unique name for the new role. For this tutorial, use **`sns_rule_role`**.

   e. Choose **Create role**.

   If you're repeating this tutorial or reusing an existing role, choose **Update role** before continuing. This updates the role's policy document to work with the SNS target.

9. Choose **Add action** and return to the **Create a rule** page.

   In the new action's tile, below **Send a message as an SNS push notification**, you can see the SNS topic that your rule will call.

   This is the only rule action you'll add to this rule.

10. To create the rule and complete this step, in **Create a rule**, scroll down to the bottom and choose **Create rule**.

Step 3: Test the AWS IoT rule and Amazon SNS notification

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the [MQTT client in the AWS IoT console](#) in a new window. This will let you edit the rule without losing the configuration of your MQTT client. If you leave the MQTT client to go to another page in the console, it won't retain any subscriptions or message logs.

To use the MQTT client to test your rule

1. In the [MQTT client in the AWS IoT console](#), subscribe to the input topics, in this case, `device/+/data`.

   a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.

   b. In **Subscription topic**, enter the topic of the input topic filter, **`device/+/data`**.

   c. Keep the rest of the fields at their default settings.

   d. Choose **Subscribe to topic**.

      In the **Subscriptions** column, under **Publish to a topic**, **`device/+/data`** appears.

2. Publish a message to the input topic with a specific device ID, **`device/32/data`**. You can't publish to MQTT topics that contain wildcard characters.

   a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.

   b. In the **Publish** field, enter the input topic name, **`device/32/data`**.

   c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{
  "temperature": 38,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

   d. Choose **Publish to topic** to publish your MQTT message.

3. Confirm that the text message was sent.

   a. In the MQTT client, under **Subscriptions**, there is a green dot next to the topic to which you subscribed earlier.

      The green dot indicates that one or more new messages have been received since the last time you looked at them.

   b. Under **Subscriptions**, choose **device/+/data** to check that the message payload matches what you just published and looks like this:

      ```
      {
        "temperature": 38,
        "humidity": 80,
        "barometer": 1013,
        "wind": {
          "velocity": 22,
          "bearing": 255
        }
      }
      ```

   c. Check the phone that you used to subscribe to the SNS topic and confirm the contents of the message payload look like this:

      ```
      {"device_id":"32","reported_temperature":38,"max_temperature":30}
      ```

      Notice that the `device_id` value is a quoted string and the `temperature` value is numeric. This is because the `topic()` function extracted the string from the input message's topic name while the `temperature` value uses the numeric value from the input message's payload.

      If you want to make the `device_id` value a numeric value, replace `topic(2)` in the rule query statement with:

      ```
      cast(topic(2) AS DECIMAL)
      ```

      Note that casting the `topic(2)` value to a numeric, `DECIMAL` value will only work if that part of the topic contains only numeric characters.

4. Try sending an MQTT message in which the temperature does not exceed the limit.

a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.

b. In the **Publish** field, enter the input topic name, **`device/33/data`**.

c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

d. To send your MQTT message, choose **Publish to topic**.

You should see the message that you sent in the **`device/+/data`** subscription. However, because the temperature value is below the max temperature in the rule query statement, you shouldn't receive a text message.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your SNS message rule

Here are some things to check, in case you're not seeing the results you expect.

- **You got an error banner**

  If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

  Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client, if you subscribed to the `device/+/data` topic filter as described in the procedure.

Things to check

- **Check the topic filter you subscribed to**

  If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

  If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

- **Check the message publish function**

  In the MQTT client, under **Subscriptions**, choose **device/+/data**, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't receive an SMS message**

  For your rule to work, it must have the correct policy that authorizes it to receive a message and send an SNS notification, and it must receive the message.

  Things to check

  - **Check the AWS Region of your MQTT client and the rule that you created**

    The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.

  - **Check that the temperature value in the message payload exceeds the test threshold**

    If the temperature value is less than or equal to 30, as defined in the rule query statement, the rule will not perform any of its actions.

  - **Check the input message topic in the rule query statement**

    For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

    Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.

  - **Check the contents of the input message payload**

For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

Check the spelling of the `temperature` field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the `temperature` field in the rule query statement must be identical to the `temperature` field in the message payload.

Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.

- **Check the republished message topic in the rule action**

  The topic to which the Republish rule action publishes the new message must match the topic to which you subscribed in the MQTT client.

  Open the rule you created in the console and check the topic to which the rule action will republish the message.

- **Check the role being used by the rule**

  The rule action must have permission to receive the original topic and publish the new topic.

  The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

  If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

In this tutorial:

- You created and tested an Amazon SNS notification topic and subscription.
- You used a simple SQL query and functions in a rule query statement to create a new message for your notification.
- You created an AWS IoT rule to send an Amazon SNS notification that used your customized message payload.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you send a few text messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the message and when it's sent. Here are some ideas to get you started.

- Change the *`device_id`* in the input message's topic and observe the effect in the text message contents.
- Change the fields selected in the rule query statement and observe the effect in the text message contents.
- Change the test in the rule query statement to test for a minimum temperature instead of a maximum temperature. Remember to change the name of `max_temperature`!
- Add a republish rule action to send an MQTT message when an SNS notification is sent.
- Try the next tutorial in this series and learn how to [Tutorial: Storing device data in a DynamoDB table](#).

Tutorial: Storing device data in a DynamoDB table

This tutorial demonstrates how to create an AWS IoT rule that sends message data to a DynamoDB table.

In this tutorial, you create a rule that sends message data from an imaginary weather sensor device to a DynamoDB table. The rule formats the data from many weather sensors such that they can be added to a single database table.

What you'll learn in this tutorial

- How to create a DynamoDB table
- How to send message data to a DynamoDB table from an AWS IoT rule
- How to use substitution templates in an AWS IoT rule
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 30 minutes to complete.

In this tutorial, you'll:

- [Step 1: Create the DynamoDB table for this tutorial](#)

- Step 2: Create an AWS IoT rule to send data to the DynamoDB table
- Step 3: Test the AWS IoT rule and DynamoDB table
- Step 4: Review the results and next steps

Before you start this tutorial, make sure that you have:

- **Set up AWS account**

  You'll need your AWS account and AWS IoT console to complete this tutorial.
- **Reviewed View MQTT messages with the AWS IoT MQTT client**

  Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.
- **Reviewed the Amazon DynamoDB overview**

  If you've not used DynamoDB before, review Getting Started with DynamoDB to become familiar with the basic concepts and operations of DynamoDB.

Step 1: Create the DynamoDB table for this tutorial

In this tutorial, you'll create a DynamoDB table with these attributes to record the data from the imaginary weather sensor devices:

- `sample_time` is a primary key and describes the time the sample was recorded.
- `device_id` is a sort key and describes the device that provided the sample
- `device_data` is the data received from the device and formatted by the rule query statement

To create the DynamoDB table for this tutorial

1. Open the DynamoDB console, and then choose **Create table**.
2. In **Create table**:
   a. In **Table name**, enter the table name: **`wx_data`**.
   b. In **Partition key**, enter **`sample_time`**, and in the option list next to the field, choose **`Number`**.
   c. In **Sort key**, enter **`device_id`**, and in the option list next to the field, choose **`Number`**.

d. At the bottom of the page, choose **Create**.

You'll define `device_data` later, when you configure the DynamoDB rule action.

Step 2: Create an AWS IoT rule to send data to the DynamoDB table

In this step, you'll use the rule query statement to format the data from the imaginary weather sensor devices to write to the database table.

A sample message payload received from a weather sensor device looks like this:

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

For the database entry, you'll use the rule query statement to flatten the structure of the message payload to look like this:

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind_velocity": 22,
  "wind_bearing": 255
}
```

In this rule, you'll also use a couple of Substitution templates. Substitution templates are expressions that let you insert dynamic values from functions and message data.

To create the AWS IoT rule to send data to the DynamoDB table

1. Open the Rules hub of the AWS IoT console. Or, you can open the AWS IoT homepage within the AWS Management Console and navigate to **Message routing>Rules**.
2. To start creating your new rule in **Rules**, choose **Create rule**.

3. In **Rule properties**:

   a. In **Rule name**, enter **`wx_data_ddb`**.

      Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.

   b. In **Rule description**, describe the rule.

      A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.

4. Choose **Next** to continue.
5. In **SQL statement**:

   a. In **SQL version**, select **`2016-03-23`**.

   b. In the **SQL statement** edit box, enter the statement:

      ```
      SELECT temperature, humidity, barometer,
        wind.velocity as wind_velocity,
        wind.bearing as wind_bearing,
      FROM 'device/+/data'
      ```

      This statement:

      - Listens for MQTT messages with a topic that matches the `device/+/data` topic filter.
      - Formats the elements of the `wind` attribute as individual attributes.
      - Passes the `temperature`, `humidity`, and `barometer` attributes unchanged.

6. Choose **Next** to continue.
7. In **Rule actions**:

   a. To open the list of rule actions for this rule, in **Action 1**, choose **DynamoDB**.

      Note

      Make sure that you choose DynamoDB and not DynamoDBv2 as the rule action.

   b. In **Table name**, choose the name of the DynamoDB table you created in a previous step: **`wx_data`**.

The **Partition key type** and **Sort key type** fields are filled with the values from your DynamoDB table.

c. In **Partition key**, enter **`sample_time`**.

d. In **Partition key value**, enter **`${timestamp()}`**.

   This is the first of the Substitution templates you'll use in this rule. Instead of using a value from the message payload, it will use the value returned from the timestamp function. To learn more, see timestamp in the *AWS IoT Core Developer Guide*.

e. In **Sort key**, enter **`device_id`**.

f. In **Sort key value**, enter **`${cast(topic(2) AS DECIMAL)}`**.

   This is the second one of the Substitution templates you'll use in this rule. It inserts the value of the second element in topic name, which is the device's ID, after it casts it to a DECIMAL value to match the numeric format of the key. To learn more about topics, see topic in the *AWS IoT Core Developer Guide*. Or to learn more about casting, see cast in the *AWS IoT Core Developer Guide*.

g. In **Write message data to this column**, enter **`device_data`**.

   This will create the `device_data` column in the DynamoDB table.

h. Leave **Operation** blank.

i. In **IAM role**, choose **Create new role**.

j. In the **Create role** dialog box, for **Role name**, enter **wx_ddb_role**. This new role will automatically contain a policy with a prefix of "aws-iot-rule" that will allow the **`wx_data_ddb`** rule to send data to the **`wx_data`** DynamoDB table you created.

k. In **IAM role**, choose **`wx_ddb_role`**.

l. At the bottom of the page, choose **Next**.

8. At the bottom of the **Review and create** page, choose **Create** to create the rule.

Step 3: Test the AWS IoT rule and DynamoDB table

To test the new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used in this test.

Open the MQTT client in the AWS IoT console in a new window. This will let you edit the rule without losing the configuration of your MQTT client. The MQTT client does not retain any subscriptions or message logs if you leave it to go to another page in the console. You'll also want a separate console window open to the DynamoDB Tables hub in the AWS IoT console to view the new entries that your rule sends.

To use the MQTT client to test your rule

1. In the MQTT client in the AWS IoT console, subscribe to the input topic, `device/+/data`.

    a. In the MQTT client, choose **Subscribe to a topic**.

    b. For **Topic filter**, enter the topic of the input topic filter, **`device/+/data`**.

    c. Choose **Subscribe**.

2. Now, publish a message to the input topic with a specific device ID, **`device/22/data`**. You can't publish to MQTT topics that contain wildcard characters.

    a. In the MQTT client, choose **Publish to a topic**.

    b. For **Topic name**, enter the input topic name, **`device/22/data`**.

    c. For **Message payload**, enter the following sample data.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

    d. To publish the MQTT message, choose **Publish**.

    e. Now, in the MQTT client, choose **Subscribe to a topic**. In the **Subscribe** column, choose the **`device/+/data`** subscription. Confirm that the sample data from the previous step appears there.

3. Check to see the row in the DynamoDB table that your rule created.

    a. In the DynamoDB Tables hub in the AWS IoT console, choose **wx_data**, and then choose the **Items** tab.

    If you're already on the **Items** tab, you might need to refresh the display by choosing the refresh icon in the upper-right corner of the table's header.

b. Notice that the **sample_time** values in the table are links and open one. If you just sent your first message, it will be the only one in the list.

   This link displays all the data in that row of the table.

c. Expand the **device_data** entry to see the data that resulted from the rule query statement.

d. Explore the different representations of the data that are available in this display. You can also edit the data in this display.

e. After you have finished reviewing this row of data, to save any changes you made, choose **Save**, or to exit without saving any changes, choose **Cancel**.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your DynamoDB rule

Here are some things to check in case you're not seeing the results you expect.

- **You got an error banner**

  If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

  Every time you publish your input message to the `device/22/data` topic, that message should appear in the MQTT client if you subscribed to the `device/+/data` topic filter as described in the procedure.

  Things to check

  - **Check the topic filter you subscribed to**

    If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

    If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

  - **Check the message publish function**

In the MQTT client, under **Subscriptions**, choose **device/+/data**, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't see your data in the DynamoDB table**

  The first thing to do is to refresh the display by choosing the refresh icon in the upper-right corner of the table's header. If that doesn't display the data you're looking for, check the following.

  Things to check

  - **Check the AWS Region of your MQTT client and the rule that you created**

    The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.
  - **Check the input message topic in the rule query statement**

    For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

    Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.
  - **Check the contents of the input message payload**

    For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

    Check the spelling of the `temperature` field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the `temperature` field in the rule query statement must be identical to the `temperature` field in the message payload.

    Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.
  - **Check the key and field names used in the rule action**

    The field names used in the topic rule must match those found in the JSON message payload of the published message.

Open the rule you created in the console and check the field names in the rule action configuration with those used in the MQTT client.

- **Check the role being used by the rule**

  The rule action must have permission to receive the original topic and publish the new topic.

  The policies that authorize the rule to receive message data and update the DynamoDB table are specific to the topics used. If you change the topic or DynamoDB table name used by the rule, you must update the rule action's role to update its policy to match.

  If you suspect this is the problem, edit the rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

After you send a few messages to the DynamoDB table with this rule, try experimenting with it to see how changing some aspects from the tutorial affect the data written to the table. Here are some ideas to get you started.

- Change the *device_id* in the input message's topic and observe the effect on the data. You could use this to simulate receiving data from multiple weather sensors.
- Change the fields selected in the rule query statement and observe the effect on the data. You could use this to filter the data stored in the table.
- Add a republish rule action to send an MQTT message for each row added to the table. You could use this for debugging.

After you have completed this tutorial, check out the section called “Formatting a notification by using an AWS Lambda function”.

Tutorial: Formatting a notification by using an AWS Lambda function

This tutorial demonstrates how to send MQTT message data to an AWS Lambda action for formatting and sending to another AWS service. In this tutorial, the AWS Lambda action uses the AWS SDK to send the formatted message to the Amazon SNS topic you created in the tutorial about how to the section called “Sending an Amazon SNS notification”.

In the tutorial about how to the section called "Sending an Amazon SNS notification", the JSON document that resulted from the rule's query statement was sent as the body of the text message. The result was a text message that looked something like this example:

```
{"device_id":"32","reported_temperature":38,"max_temperature":30}
```

In this tutorial, you'll use an AWS Lambda rule action to call an AWS Lambda function that formats the data from the rule query statement into a friendlier format, such as this example:

```
Device 32 reports a temperature of 38, which exceeds the limit of 30.
```

The AWS Lambda function you'll create in this tutorial formats the message string by using the data from the rule query statement and calls the SNS publish function of the AWS SDK to create the notification.

What you'll learn in this tutorial

- How to create and test an AWS Lambda function
- How to use the AWS SDK in an AWS Lambda function to publish an Amazon SNS notification
- How to use simple SQL queries and functions in a rule query statement
- How to use the MQTT client to test an AWS IoT rule

This tutorial takes about 45 minutes to complete.

In this tutorial, you'll:

- Step 1: Create an AWS Lambda function that sends a text message
- Step 2: Create an AWS IoT rule with an AWS Lambda rule action
- Step 3: Test the AWS IoT rule and AWS Lambda rule action
- Step 4: Review the results and next steps

Before you start this tutorial, make sure that you have:

- **Set up AWS account**

  You'll need your AWS account and AWS IoT console to complete this tutorial.
- **Reviewed View MQTT messages with the AWS IoT MQTT client**

Be sure you can use the MQTT client to subscribe and publish to a topic. You'll use the MQTT client to test your new rule in this procedure.

- **Completed the other rules tutorials in this section**

  This tutorial requires the SNS notification topic you created in the tutorial about how to the section called "Sending an Amazon SNS notification". It also assumes that you've completed the other rules-related tutorials in this section.

- **Reviewed the AWS Lambda overview**

  If you haven't used AWS Lambda before, review AWS Lambda and Getting started with Lambda to learn its terms and concepts.

Step 1: Create an AWS Lambda function that sends a text message

The AWS Lambda function in this tutorial receives the result of the rule query statement, inserts the elements into a text string, and sends the resulting string to Amazon SNS as the message in a notification.

Unlike the tutorial about how to the section called "Sending an Amazon SNS notification", which used an AWS IoT rule action to send the notification, this tutorial sends the notification from the Lambda function by using a function of the AWS SDK. The actual Amazon SNS notification topic used in this tutorial, however, is the same one that you used in the tutorial about how to the section called "Sending an Amazon SNS notification".

To create an AWS Lambda function that sends a text message

1. Create a new AWS Lambda function.

   a. In the AWS Lambda console, choose **Create function**.

   b. In **Create function**, select **Use a blueprint**.

      Search for and select the **`hello-world-python`** blueprint, and then choose **Configure**.

   c. In **Basic information**:

      i. In **Function name**, enter the name of this function, **`format-high-temp-notification`**.

      ii. In **Execution role**, choose **Create a new role from AWS policy templates**.

iii. In Role name, enter the name of the new role, **`format-high-temp-notification-role`**.

iv. In **Policy templates - *optional***, search for and select **Amazon SNS publish policy**.

v. Choose **Create function**.

2. Modify the blueprint code to format and send an Amazon SNS notification.

   a. After you created your function, you should see the **format-high-temp-notification** details page. If you don't, open it from the [Lambda **Functions**] page.

   b. In the **format-high-temp-notification** details page, choose the **Configuration** tab and scroll to the **Function code** panel.

   c. In the **Function code** window, in the **Environment** pane, choose the Python file, `lambda_function.py`.

   d. In the **Function code** window, delete all of the original program code from the blueprint and replace it with this code.

```
import boto3
#
#   expects event parameter to contain:
#   {
#       "device_id": "32",
#       "reported_temperature": 38,
#       "max_temperature": 30,
#       "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"
#   }
#
#   sends a plain text string to be used in a text message
#
#      "Device {0} reports a temperature of {1}, which exceeds the limit of {2}."
#
#   where:
#       {0} is the device_id value
#       {1} is the reported_temperature value
#       {2} is the max_temperature value
#
def lambda_handler(event, context):

    # Create an SNS client to send notification
    sns = boto3.client('sns')
```

```
    # Format text message from data
    message_text = "Device {0} reports a temperature of {1}, which exceeds the
 limit of {2}.".format(
            str(event['device_id']),
            str(event['reported_temperature']),
            str(event['max_temperature'])
        )

    # Publish the formatted message
    response = sns.publish(
            TopicArn = event['notify_topic_arn'],
            Message = message_text
        )

    return response
```

   e. Choose **Deploy**.

3. In a new window, look up the Amazon Resource Name (ARN) of your Amazon SNS topic from the tutorial about how to [the section called “Sending an Amazon SNS notification”](#).

   a. In a new window, open the [Topics page of the Amazon SNS console](#).
   b. In the **Topics** page, find the **high_temp_notice** notification topic in the list of Amazon SNS topics.
   c. Find the **ARN** of the **high_temp_notice** notification topic to use in the next step.

4. Create a test case for your Lambda function.

   a. In the [Lambda **Functions**](#) page of the console, on the **format-high-temp-notification** details page, choose **Select a test event** in the upper right corner of the page (even though it looks disabled), and then choose **Configure test events**.
   b. In **Configure test event**, choose **Create new test event**.
   c. In **Event name**, enter **`SampleRuleOutput`**.
   d. In the JSON editor below **Event name**, paste this sample JSON document. This is an example of what your AWS IoT rule will send to the Lambda function.

```
{
  "device_id": "32",
  "reported_temperature": 38,
  "max_temperature": 30,
  "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"
```

```
}
```

e. Refer to the window that has the **ARN** of the **high_temp_notice** notification topic and copy the ARN value.

f. Replace the `notify_topic_arn` value in the JSON editor with the ARN from your notification topic.

   Keep this window open so you can use this ARN value again when you create the AWS IoT rule.

g. Choose **Create**.

5. Test the function with sample data.

   a. In the **format-high-temp-notification** details page, in the upper-right corner of the page, confirm that **SampleRuleOutput** appears next to the **Test** button. If it doesn't, choose it from the list of available test events.

   b. To send the sample rule output message to your function, choose **Test**.

If the function and the notification both worked, you will get a text message on the phone that subscribed to the notification.

If you didn't get a text message on the phone, check the result of the operation. In the **Function code** panel, in the **Execution result** tab, review the response to find any errors that occurred. Don't continue to the next step until your function can send the notification to your phone.

Step 2: Create an AWS IoT rule with an AWS Lambda rule action

In this step, you'll use the rule query statement to format the data from the imaginary weather sensor device to send to a Lambda function, which will format and send a text message.

A sample message payload received from the weather devices looks like this:

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
```

```
}
```

In this rule, you'll use the rule query statement to create a message payload for the Lambda function that looks like this:

```
{
  "device_id": "32",
  "reported_temperature": 38,
  "max_temperature": 30,
  "notify_topic_arn": "arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice"
}
```

This contains all the information the Lambda function needs to format and send the correct text message.

To create the AWS IoT rule to call a Lambda function

1. Open the **Rules** hub of the AWS IoT console.
2. To start creating your new rule in **Rules**, choose **Create**.
3. In the top part of **Create a rule**:

   a. In **Name**, enter the rule's name, **`wx_friendly_text`**.

      Remember that a rule name must be unique within your AWS account and Region, and it can't have any spaces. We've used an underscore character in this name to separate the two words in the rule's name.

   b. In **Description**, describe the rule.

      A meaningful description makes it easier to remember what this rule does and why you created it. The description can be as long as needed, so be as detailed as possible.

4. In **Rule query statement** of **Create a rule**:

   a. In **Using SQL version**, select **`2016-03-23`**.

   b. In the **Rule query statement** edit box, enter the statement:

      ```
      SELECT
        cast(topic(2) AS DECIMAL) as device_id,
        temperature as reported_temperature,
        30 as max_temperature,
        'arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice' as notify_topic_arn
      ```

```
FROM 'device/+/data' WHERE temperature > 30
```

This statement:

- Listens for MQTT messages with a topic that matches the `device/+/data` topic filter and that have a `temperature` value greater than 30.
- Selects the second element from the topic string, converts it to a decimal number, and then assigns it to the `device_id` field.
- Selects the value of the `temperature` field from the message payload and assigns it to the `reported_temperature` field.
- Creates a constant value, `30`, to represent the limit value and assigns it to the `max_temperature` field.
- Creates a constant value for the `notify_topic_arn` field.

c. Refer to the window that has the **ARN** of the **high_temp_notice** notification topic and copy the ARN value.

d. Replace the ARN value (*`arn:aws:sns:us-east-1:57EXAMPLE833:high_temp_notice`*) in the rule query statement editor with the ARN of your notification topic.

5. In **Set one or more actions**:

   a. To open up the list of rule actions for this rule, choose **Add action**.

   b. In **Select an action**, choose **Send a message to a Lambda function**.

   c. To open the selected action's configuration page, at the bottom of the action list, choose **Configure action**.

6. In **Configure action**:

   a. In **Function name**, choose **Select**.

   b. Choose **format-high-temp-notification**.

   c. At the bottom of **Configure action**, choose **Add action**.

   d. To create the rule, at the bottom of **Create a rule**, choose **Create rule**.

Step 3: Test the AWS IoT rule and AWS Lambda rule action

To test your new rule, you'll use the MQTT client to publish and subscribe to the MQTT messages used by this rule.

Open the MQTT client in the AWS IoT console in a new window. Now you can edit the rule without losing the configuration of your MQTT client. If you leave the MQTT client to go to another page in the console, you'll lose your subscriptions or message logs.

To use the MQTT client to test your rule

1. In the MQTT client in the AWS IoT console, subscribe to the input topics, in this case, `device/+/data`.

    a. In the MQTT client, under **Subscriptions**, choose **Subscribe to a topic**.

    b. In **Subscription topic**, enter the topic of the input topic filter, **`device/+/data`**.

    c. Keep the rest of the fields at their default settings.

    d. Choose **Subscribe to topic**.

        In the **Subscriptions** column, under **Publish to a topic**, **`device/+/data`** appears.

2. Publish a message to the input topic with a specific device ID, **`device/32/data`**. You can't publish to MQTT topics that contain wildcard characters.

    a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.

    b. In the **Publish** field, enter the input topic name, **`device/32/data`**.

    c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

    ```
    {
      "temperature": 38,
      "humidity": 80,
      "barometer": 1013,
      "wind": {
        "velocity": 22,
        "bearing": 255
      }
    }
    ```

    d. To publish your MQTT message, choose **Publish to topic**.

3. Confirm that the text message was sent.

    a. In the MQTT client, under **Subscriptions**, there is a green dot next to the topic to which you subscribed earlier.

The green dot indicates that one or more new messages have been received since the last time you looked at them.

b. Under **Subscriptions**, choose **device/+/data** to check that the message payload matches what you just published and looks like this:

```
{
  "temperature": 38,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
}
```

c. Check the phone that you used to subscribe to the SNS topic and confirm the contents of the message payload look like this:

```
Device 32 reports a temperature of 38, which exceeds the limit of 30.
```

If you change the topic ID element in the message topic, remember that casting the `topic(2)` value to a numeric value will only work if that element in the message topic contains only numeric characters.

4. Try sending an MQTT message in which the temperature does not exceed the limit.

a. In the MQTT client, under **Subscriptions**, choose **Publish to topic**.

b. In the **Publish** field, enter the input topic name, **`device/33/data`**.

c. Copy the sample data shown here and, in the edit box below the topic name, paste the sample data.

```
{
  "temperature": 28,
  "humidity": 80,
  "barometer": 1013,
  "wind": {
    "velocity": 22,
    "bearing": 255
  }
```

```
}
```

d. To send your MQTT message, choose **Publish to topic**.

You should see the message that you sent in the **device/+/data** subscription; however, because the temperature value is below the max temperature in the rule query statement, you shouldn't receive a text message.

If you don't see the correct behavior, check the troubleshooting tips.

Troubleshooting your AWS Lambda rule and notification

Here are some things to check, in case you're not seeing the results you expect.

- **You got an error banner**

  If an error appeared when you published the input message, correct that error first. The following steps might help you correct that error.

- **You don't see the input message in the MQTT client**

  Every time you publish your input message to the `device/32/data` topic, that message should appear in the MQTT client, if you subscribed to the `device/+/data` topic filter as described in the procedure.

  Things to check

  - **Check the topic filter you subscribed to**

    If you subscribed to the input message topic as described in the procedure, you should see a copy of the input message every time you publish it.

    If you don't see the message, check the topic name you subscribed to and compare it to the topic to which you published. Topic names are case sensitive and the topic to which you subscribed must be identical to the topic to which you published the message payload.

  - **Check the message publish function**

    In the MQTT client, under **Subscriptions**, choose **device/+/data**, check the topic of the publish message, and then choose **Publish to topic**. You should see the message payload from the edit box below the topic appear in the message list.

- **You don't receive an SMS message**

  For your rule to work, it must have the correct policy that authorizes it to receive a message and send an SNS notification, and it must receive the message.

  Things to check

  - **Check the AWS Region of your MQTT client and the rule that you created**

    The console in which you're running the MQTT client must be in the same AWS Region as the rule you created.
  - **Check that the temperature value in the message payload exceeds the test threshold**

    If the temperature value is less than or equal to 30, as defined in the rule query statement, the rule will not perform any of its actions.
  - **Check the input message topic in the rule query statement**

    For the rule to work, it must receive a message with the topic name that matches the topic filter in the FROM clause of the rule query statement.

    Check the spelling of the topic filter in the rule query statement with that of the topic in the MQTT client. Topic names are case sensitive and the message's topic must match the topic filter in the rule query statement.
  - **Check the contents of the input message payload**

    For the rule to work, it must find the data field in the message payload that is declared in the SELECT statement.

    Check the spelling of the `temperature` field in the rule query statement with that of the message payload in the MQTT client. Field names are case sensitive and the `temperature` field in the rule query statement must be identical to the `temperature` field in the message payload.

    Make sure that the JSON document in the message payload is correctly formatted. If the JSON has any errors, such as a missing comma, the rule will not be able to read it.
  - **Check the Amazon SNS notification**

    In Step 1: Create an Amazon SNS topic that sends a SMS text message, refer to step 3 that describes how to test the Amazon SNS notification and test the notification to make sure the notification works.

- **Check the Lambda function**

  In Step 1: Create an AWS Lambda function that sends a text message, refer to step 5 that describes how to test the Lambda function using test data and test the Lambda function.

- **Check the role being used by the rule**

  The rule action must have permission to receive the original topic and publish the new topic.

  The policies that authorize the rule to receive message data and republish it are specific to the topics used. If you change the topic used to republish the message data, you must update the rule action's role to update its policy to match the current topic.

  If you suspect this is the problem, edit the Republish rule action and create a new role. New roles created by the rule action receive the authorizations necessary to perform these actions.

Step 4: Review the results and next steps

In this tutorial:

- You created an AWS IoT rule to call a Lambda function that sent an Amazon SNS notification that used your customized message payload.
- You used a simple SQL query and functions in a rule query statement to create a new message payload for your Lambda function.
- You used the MQTT client to test your AWS IoT rule.

Next steps

After you send a few text messages with this rule, try experimenting with it to see how changing some aspects of the tutorial affect the message and when it's sent. Here are some ideas to get you started.

- Change the *`device_id`* in the input message's topic and observe the effect in the text message contents.
- Change the fields selected in the rule query statement, update the Lambda function to use them in a new message, and observe the effect in the text message contents.
- Change the test in the rule query statement to test for a minimum temperature instead of a maximum temperature. Update the Lambda function to format a new message and remember to change the name of `max_temperature`.

- To learn more about how to find errors that might occur while you're developing and using AWS IoT rules, see Monitoring AWS IoT.

Retaining device state while the device is offline with Device Shadows

These tutorials show you how to use the AWS IoT Device Shadow service to store and update the state information of a device. The Shadow document, which is a JSON document, shows the change in the device's state based on the messages published by a device, local app, or service. In this tutorial, the Shadow document shows the change in the color of a light bulb. These tutorials also show how the shadow stores this information even when the device is disconnected from the internet, and passes the latest state information back to the device when it comes back online and requests this information.

We recommend that you try these tutorials in the order they're shown here, starting with the AWS IoT resources you need to create and the necessary hardware setup, which also helps you learn the concepts incrementally. These tutorials show how to configure and connect a Raspberry Pi device for use with AWS IoT. If you don't have the required hardware, you can follow these tutorials by adapting them to a device of your choice or by creating a virtual device with Amazon EC2.

Tutorial scenario overview

The scenario for these tutorials is a local app or service that changes the color of a light bulb and that publishes its data to reserved shadow topics. These tutorials are similar to the Device Shadow functionality described in the interactive getting started tutorial and are implemented on a Raspberry Pi device. The tutorials in this section focus on a single, classic shadow while showing how you might accommodate named shadows or multiple devices.

The following tutorials will help you learn how to use the AWS IoT Device Shadow service.

- **Tutorial: Preparing your Raspberry Pi to run the shadow application**

  This tutorial shows how to set up a Raspberry Pi device for connecting with AWS IoT. You'll also create an AWS IoT policy document and a thing resource, download the certificates, and then attach the policy to that thing resource. This tutorial takes about 30 minutes to complete.
- **Tutorial: Installing the Device SDK and running the sample application for Device Shadows**

  This tutorial shows how to install the required tools, software, and the AWS IoT Device SDK for Python, and then run the sample shadow application. This tutorial builds on concepts presented in Connect a Raspberry Pi or other device and takes 20 minutes to complete.

- **[Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client](#)**

  This tutorial shows how you use the `shadow.py` sample app and **AWS IoT console** to observe the interaction between AWS IoT Device Shadows and the state changes of the light bulb. The tutorial also shows how to send MQTT messages to the Device Shadow's reserved topics. This tutorial can take 45 minutes to complete.

AWS IoT Device Shadow overview

A Device Shadow is a persistent, virtual representation of a device that is managed by a thing resource you create in the AWS IoT registry. The Shadow document is a JSON or a JavaScript notation doc that is used to store and retrieve the current state information for a device. You can use the shadow to get and set the state of a device over MQTT topics or HTTP REST APIs, regardless of whether the device is connected to the internet.

A Shadow document contains a `state` property that describes these aspects of the device's state.

- `desired`: Apps specify the desired states of device properties by updating the `desired` object.
- `reported`: Devices report their current state in the `reported` object.
- `delta`: AWS IoT reports differences between the desired and the reported state in the `delta` object.

Here is an example of a Shadow state document.

```
{
  "state": {
    "desired": {
      "color": "green"
      },
    "reported": {
      "color": "blue"
      },
    "delta": {
      "color": "green"
      }
   }
}
```

To update a device's Shadow document, you can use the reserved MQTT topics, the Device Shadow REST APIs that support the GET, UPDATE, and DELETE operations with HTTP, and the AWS IoT CLI.

In the previous example, say you want to change the `desired` color to `yellow`. To do this, send a request to the UpdateThingShadow API or publish a message to the Update topic, `$aws/things/THING_NAME/shadow/update`.

```
{
  "state": {
    "desired": {
      "color": yellow
    }
  }
}
```

Updates affect only the fields specified in the request. After successfully updating the Device Shadow, AWS IoT publishes the new `desired` state to the `delta` topic, `$aws/things/THING_NAME/shadow/delta`. The Shadow document in this case looks like this:

```
{
  "state": {
    "desired": {
      "color": yellow
    },
    "reported": {
      "color": green
    },
    "delta": {
      "color": yellow
      }
  }
}
```

The new state is then reported to the AWS IoT Device Shadow using the `Update` topic `$aws/things/THING_NAME/shadow/update` with the following JSON message:

```
{
  "state": {
    "reported": {
      "color": yellow
    }
```

```
    }
}
```

If you want to get the current state information, send a request to the GetThingShadow API or publish an MQTT message to the Get topic, `$aws/things/THING_NAME/shadow/get`.

For more information about using the Device Shadow service, see AWS IoT Device Shadow service.

For more information about using Device Shadows in devices, apps, and services, see Using shadows in devices and Using shadows in apps and services.

For information about interacting with AWS IoT shadows, see Interacting with shadows.

For information about the MQTT reserved topics and HTTP REST APIs, see Device Shadow MQTT topics and Device Shadow REST API.

Tutorial: Preparing your Raspberry Pi to run the shadow application

This tutorial demonstrates how to set up and configure a Raspberry Pi device and create the AWS IoT resources that a device requires to connect and exchange MQTT messages.

> **Note**
>
> If you're planning to the section called "Create a virtual device with Amazon EC2", you can skip this page and continue to the section called "Configure your device". You'll create these resources when you create your virtual thing. If you would like to use a different device instead of the Raspberry Pi, you can try to follow these tutorials by adapting them to a device of your choice.

In this tutorial, you'll learn how to:

- Set up a Raspberry Pi device and configure it for use with AWS IoT.
- Create an AWS IoT policy document, which authorizes your device to interact with AWS IoT services.
- Create a thing resource in AWS IoT the X.509 device certificates, and then attach the policy document.

The thing is the virtual representation of your device in the AWS IoT registry. The certificate authenticates your device to AWS IoT Core, and the policy document authorizes your device to interact with AWS IoT.

How to run this tutorial

To run the `shadow.py` sample application for Device Shadows, you'll need a Raspberry Pi device that connects to AWS IoT. We recommend that you follow this tutorial in the order it's presented here, starting with setting up the Raspberry Pi and it's accessories, and then creating a policy and attaching the policy to a thing resource that you create. You can then follow this tutorial by using the graphical user interface (GUI) supported by the Raspberry Pi to open the AWS IoT console on the device's web browser, which also makes it easier to download the certificates directly to your Raspberry Pi for connecting to AWS IoT.

Before you start this tutorial, make sure that you have:

- An AWS account. If you don't have one, complete the steps described in Set up AWS account before you continue. You'll need your AWS account and AWS IoT console to complete this tutorial.
- The Raspberry Pi and its necessary accessories. You'll need:
  - A Raspberry Pi 3 Model B or more recent model. This tutorial might work on earlier versions of the Raspberry Pi, but we haven't tested it.
  - Raspberry Pi OS (32-bit) or later. We recommend using the latest version of the Raspberry Pi OS. Earlier versions of the OS might work, but we haven't tested it.
  - An Ethernet or Wi-Fi connection.
  - Keyboard, mouse, monitor, cables, and power supplies.

This tutorial takes about 30 minutes to complete.

Step 1: Set up and configure Raspberry Pi device

In this section, we'll configure a Raspberry Pi device for use with AWS IoT.

> **Important**
>
> Adapting these instructions to other devices and operating systems can be challenging. You'll need to understand your device well enough to be able to interpret these instructions

and apply them to your device. If you encounter difficulties, you might try one of the other device options as an alternative, such as Create a virtual device with Amazon EC2 or Use your Windows or Linux PC or Mac as an AWS IoT device.

You'll need to configure your Raspberry Pi such that it can start the operating system (OS), connect to the internet, and allow you to interact with it at a command line interface. You can also use the graphical user interface (GUI) supported with the Raspberry Pi to open the AWS IoT console and run the rest of this tutorial.

To set up the Raspberry Pi

1. Insert the SD card into the MicroSD card slot on the Raspberry Pi. Some SD cards come pre-loaded with an installation manager that prompts you with a menu to install the OS after booting up the board. You can also use the Raspberry Pi imager to install the OS on your card.
2. Connect an HDMI TV or monitor to the HDMI cable that connects to the HDMI port of the Raspberry Pi.
3. Connect the keyboard and mouse to the USB ports of the Raspberry Pi and then plug in the power adapter to boot up the board.

After the Raspberry Pi boots up, if the SD card came pre-loaded with the installation manager, a menu appears to install the operating system. If you have trouble installing the OS, you can try the following steps. For more information about setting up the Raspberry Pi, see Setting up your Raspberry Pi.

If you're having trouble setting up the Raspberry Pi:

- Check whether you inserted the SD card before booting up the board. If you plug in the SD card after booting up the board, the installation menu might not appear.
- Make sure that the TV or monitor is turned on and the correct input is selected.
- Ensure that you are using Raspberry Pi compatible software.

After you have installed and configured the Raspberry Pi OS, open the Raspberry Pi's web browser and navigate to the AWS IoT Core console to continue the rest of the steps in this tutorial.

If you can open the AWS IoT Core console, you're Raspberry Pi is ready and you can continue to the section called “Provisioning your device in AWS IoT ”.

If you're having trouble or need additional help, see Getting help for your Raspberry Pi.

Tutorial: Provisioning your device in AWS IoT

This section creates the AWS IoT Core resources that your tutorial will use.

Steps to provision your device in AWS IoT

Step 1: Create an AWS IoT policy for the Device Shadow

X.509 certificates authenticate your device with AWS IoT Core. AWS IoT policies are attached to the certificate that permits the device to perform AWS IoT operations, such as subscribing or publishing to MQTT reserved topics used by the Device Shadow service. Your device presents its certificate when it connects and sends messages to AWS IoT Core.

In this procedure, you'll create a policy that allows your device to perform the AWS IoT operations necessary to run the example program. We recommend that you create a policy that grants only the permissions required to perform the task. You create the AWS IoT policy first, and then attach it to the device certificate that you'll create later.

To create an AWS IoT policy

1. On the left menu, choose **Secure**, and then choose **Policies**. If your account has existing policies, choose **Create**, otherwise, on the **You don’t have a policy yet** page, choose **Create a policy**.
2. On the **Create a policy** page:

   a. Enter a name for the policy in the **Name** field (for example, **`My_Device_Shadow_policy`**). Do not use personally identifiable information in your policy names.

   b. In the policy document, you describe connect, subscribe, receive, and publish actions that give the device permission to publish and subscribe to the MQTT reserved topics.

      Copy the following sample policy and paste it in your policy document. Replace `thingname` with the name of the thing that you'll create (for example, `My_light_bulb`), `region` with the AWS IoT Region where you're using the services, and `account` with your AWS account number. For more information about AWS IoT policies, see [AWS IoT Core policies](#).

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/get",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/update"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/get/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/get/rejected",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/update/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/update/rejected",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/
things/thingname/shadow/update/delta"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
```

```
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingname/shadow/get/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingname/shadow/get/rejected",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingname/shadow/update/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingname/shadow/update/rejected",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingname/shadow/update/delta"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/test-*"
        }
    ]
}
```

Step 2: Create a thing resource and attach the policy to the thing

Devices connected to AWS IoT can be represented by *thing resources* in the AWS IoT registry. A *thing resource* represents a specific device or logical entity, such as the light bulb in this tutorial.

To learn how to create a thing in AWS IoT, follow the steps described in Create a thing object. Here are some key things to note as you follow the steps in that tutorial:

1. Choose **Create a single thing**, and in the **Name** field, enter a name for the thing that is the same as the `thingname` (for example, `My_light_bulb`) you specified when you created the policy earlier.

   You can't change a thing name after it has been created. If you gave it a different name other than `thingname`, create a new thing with name as `thingname` and delete the old thing.

   > **Note**
   >
   > Do not use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.

2. We recommend that you download each of the certificate files on the **Certificate created!** page into a location where you can easily find them. You'll need to install these files for running the sample application.

   We recommend that you download the files into a `certs` subdirectory in your home directory on the Raspberry Pi and name each of them with a simpler name as suggested in the following table.

   Certificate file names

File	File path
Root CA certificate	`~/certs/Amazon-root-CA-1.pem`
Device certificate	`~/certs/device.pem.crt`
Private key	`~/certs/private.pem.key`

3. After you activate the certificate to enable connections to AWS IoT, choose **Attach a policy** and make sure you attach the policy that you created earlier (for example, **`My_Device_Shadow_policy`**) to the thing.

   After you've created a thing, you can see your thing resource displayed in the list of things in the AWS IoT console.

Step 3: Review the results and next steps

In this tutorial, you learned how to:

- Set up and configure the Raspberry Pi device.
- Create an AWS IoT policy document that authorizes your device to interact with AWS IoT services.
- Create a thing resource and associated X.509 device certificate, and attach the policy document to it.

Next steps

You can now install the AWS IoT device SDK for Python, run the `shadow.py` sample application, and use Device Shadows to control the state. For more information about how to run this tutorial, see Tutorial: Installing the Device SDK and running the sample application for Device Shadows.

Tutorial: Installing the Device SDK and running the sample application for Device Shadows

This section shows how you can install the required software and the AWS IoT Device SDK for Python and run the `shadow.py` sample application to edit the Shadow document and control the shadow's state.

In this tutorial, you'll learn how to:

- Use the installed software and AWS IoT Device SDK for Python to run the sample app.
- Learn how entering a value using the sample app publishes the desired value in the AWS IoT console.
- Review the `shadow.py` sample app and how it uses the MQTT protocol to update the shadow's state.

Before you run this tutorial:

You must have set up your AWS account, configured your Raspberry Pi device, and created an AWS IoT thing and policy that gives the device permissions to publish and subscribe to the MQTT reserved topics of the Device Shadow service. For more information, see Tutorial: Preparing your Raspberry Pi to run the shadow application.

You must have also installed Git, Python, and the AWS IoT Device SDK for Python. This tutorial builds on the concepts presented in the tutorial Connect a Raspberry Pi or other device. If you haven't tried that tutorial, we recommend that you follow the steps described in that tutorial to install the certificate files and Device SDK and then come back to this tutorial to run the `shadow.py` sample app.

In this tutorial, you'll:

- Step 1: Run the shadow.py sample app
- Step 2: Review the shadow.py Device SDK sample app
- Step 3: Troubleshoot problems with the shadow.py sample app
- Step 4: Review the results and next steps

This tutorial takes about 20 minutes to complete.

Step 1: Run the shadow.py sample app

Before you run the `shadow.py` sample app, you'll need the following information in addition to the names and location of the certificate files that you installed.

Application parameter values

Parameter	Where to find the value
`your-iot-thing-name`	Name of the AWS IoT thing that you created earlier in the section called "Step 2: Create a thing resource and attach the policy to the thing". To find this value, in the AWS IoT console, choose **Manage**, and then choose **Things**.
`your-iot-endpoint`	The *`your-iot-endpoint`* value has a format of: *`endpoint_id`*`-ats.iot.`*`region`*`.amazonaws.com`, for example, `a3qj468EXAMPLE-ats.iot.us-west-2.amazonaws.com`. To find this value: 1. In the AWS IoT console, choose **Manage**, and then choose **Things**. 2. Choose the IoT thing you created for your device, **My_light_bulb**, that you used earlier, and then choose **Interact**. On the thing details page, your endpoint is displayed in the **HTTPS** section.

Install and run the sample app

1. Navigate to the sample app directory.

```
cd ~/aws-iot-device-sdk-python-v2/samples/service-clients
```

2. In the command line window, replace *your-iot-endpoint* and *your-iot-thing-name* as indicated and run this command.

```
python3 shadow.py --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/certs/
device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint --
thing_name your-iot-thing-name
```

3. Observe that the sample app:

   1. Connects to the AWS IoT service for your account.
   2. Subscribes to `Delta` events and `Update` and `Get` responses.
   3. Prompts you to enter a desired value in the terminal.
   4. Displays output similar to the following:

```
Connecting to a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com with client ID
 'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...
Connected!
Subscribing to Delta events...
Subscribing to Update responses...
Subscribing to Get responses...
Requesting current shadow state...
Launching thread to read user input...
Finished getting initial shadow state.
Shadow contains reported value 'off'.
Enter desired value:
```

Note

If you're having trouble running the `shadow.py` sample app, review the section called "Step 3: Troubleshoot problems with the `shadow.py` sample app". To get additional information that might help you correct the problem, add the `--verbosity debug` parameter to the command line so the sample app displays detailed messages about what it's doing.

Enter values and observe the updates in Shadow document

You can enter values in the terminal to specify the `desired` value, which also updates the `reported` value. Say you enter the color `yellow` in the terminal. The `reported` value is also updated to the color `yellow`. The following shows the messages displayed in the terminal:

```
Enter desired value:
yellow
Changed local shadow value to 'yellow'.
Updating reported shadow value to 'yellow'...
Update request published.
Finished updating reported shadow value to 'yellow'.
```

When you publish this update request, AWS IoT creates a default, classic shadow for the thing resource. You can observe the update request that you published to the `reported` and `desired` values in the AWS IoT console by looking at the Shadow document for the thing resource that you created (for example, `My_light_bulb`). To see the update in the Shadow document:

1. In the AWS IoT console, choose **Manage** and then choose **Things**.
2. In the list of things displayed, select the thing that you created, choose **Shadows**, and then choose **Classic Shadow**.

The Shadow document should look similar to the following, showing the `reported` and `desired` values set to the color `yellow`. You see these values in the **Shadow state** section of the document.

```
{
"desired": {
  "welcome": "aws-iot",
  "color": "yellow"
},
"reported": {
  "welcome": "aws-iot",
  "color": "yellow"
}
}
```

You also see a **Metadata** section that contains the timestamp information and version number of the request.

You can use the state document version to ensure you are updating the most recent version of a device's Shadow document. If you send another update request, the version number increments by 1. When you supply a version with an update request, the service rejects the request with an HTTP

409 conflict response code if the current version of the state document doesn't match the version supplied.

```
{
"metadata": {
  "desired": {
    "welcome": {
      "timestamp": 1620156892
    },
    "color": {
      "timestamp": 1620156893
    }
  },
  "reported": {
    "welcome": {
      "timestamp": 1620156892
    },
    "color": {
      "timestamp": 1620156893
    }
  }
},
"version": 10
}
```

To learn more about the Shadow document and observe changes to the state information, proceed to the next tutorial Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client as described in the Step 4: Review the results and next steps section of this tutorial. Optionally, you can also learn about the `shadow.py` sample code and how it uses the MQTT protocol in the following section.

Step 2: Review the shadow.py Device SDK sample app

This section reviews the `shadow.py` sample app from the **AWS IoT Device SDK v2 for Python** used in this tutorial. Here, we'll review how it connects to AWS IoT Core by using the MQTT and MQTT over WSS protocol. The AWS common runtime (AWS-CRT) library provides the low-level communication protocol support and is included with the AWS IoT Device SDK v2 for Python.

While this tutorial uses MQTT and MQTT over WSS, AWS IoT supports devices that publish HTTPS requests. For an example of a Python program that sends an HTTP message from a device, see the HTTPS code example using Python’s `requests` library.

For information about how you can make an informed decision about which protocol to use for your device communications, review the [Choosing an application protocol for your device communication](#).

MQTT

The `shadow.py` sample calls `mtls_from_path` (shown here) in the [`mqtt_connection_builder`](#) to establish a connection with AWS IoT Core by using the MQTT protocol. `mtls_from_path` uses X.509 certificates and TLS v1.2 to authenticate the device. The AWS-CRT library handles the lower-level details of that connection.

```
mqtt_connection = mqtt_connection_builder.mtls_from_path(
  endpoint=args.endpoint,
  cert_filepath=args.cert,
  pri_key_filepath=args.key,
  ca_filepath=args.ca_file,
  client_bootstrap=client_bootstrap,
  on_connection_interrupted=on_connection_interrupted,
  on_connection_resumed=on_connection_resumed,
  client_id=args.client_id,
  clean_session=False,
  keep_alive_secs=6
)
```

- `endpoint` is your AWS IoT endpoint that you passed in from the command line and `client_id` is the ID that uniquely identifies this device in the AWS Region.
- `cert_filepath`, `pri_key_filepath`, and `ca_filepath` are the paths to the device's certificate and private key files, and the root CA file.
- `client_bootstrap` is the common runtime object that handles socket communication activities, and is instantiated prior to the call to `mqtt_connection_builder.mtls_from_path`.
- `on_connection_interrupted` and `on_connection_resumed` are callback functions to call when the device's connection is interrupted and resumed.
- `clean_session` is whether to start a new, persistent session, or if one is present, reconnect to an existing one. `keep_alive_secs` is the keep alive value, in seconds, to send in the `CONNECT` request. A ping will automatically be sent at this interval. The server assumes that the connection is lost if it doesn't receive a ping after 1.5 times this value.

The `shadow.py` sample also calls `websockets_with_default_aws_signing` in the [`mqtt_connection_builder`](mqtt_connection_builder) to establish a connection with AWS IoT Core using MQTT protocol over WSS. MQTT over WSS also uses the same parameters as MQTT and takes these additional parameters:

- `region` is the AWS signing Region used by Signature V4 authentication, and `credentials_provider` is the AWS credentials provided to use for authentication. The Region is passed from the command line, and the `credentials_provider` object is instantiated just prior to the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.
- `websocket_proxy_options` is the HTTP proxy options, if using a proxy host. In the `shadow.py` sample app, this value is instantiated just prior to the call to `mqtt_connection_builder.websockets_with_default_aws_signing`.

Subscribe to Shadow topics and events

The `shadow.py` sample attempts to establish a connection and waits to be fully connected. If it's not connected, commands are queued up. Once connected, the sample subscribes to delta events and update and get messages, and publishes messages with a Quality of Service (QoS) level of 1 (`mqtt.QoS.AT_LEAST_ONCE`).

When a device subscribes to a message with QoS level 1, the message broker saves the messages that the device is subscribed to until they can be sent to the device. The message broker resends the messages until it receives a `PUBACK` response from the device.

For more information about the MQTT protocol, see Review the MQTT protocol and MQTT.

For more information about how MQTT, MQTT over WSS, persistent sessions, and QoS levels that are used in this tutorial, see Review the pubsub.py Device SDK sample app.

Step 3: Troubleshoot problems with the `shadow.py` sample app

When you run the `shadow.py` sample app, you should see some messages displayed in the terminal and a prompt to enter a `desired` value. If the program throws an error, then to debug the error, you can start by checking whether you ran the correct command for your system.

In some cases, the error message might indicate connection issues and look similar to: `Host name was invalid for dns resolution` or `Connection was closed unexpectedly`. In such cases, here are some things you can check:

- **Check the endpoint address in the command**

  Review the `endpoint` argument in the command you entered to run the sample app, (for example, `a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com`) and check this value in the **AWS IoT console**.

  To check whether you used the correct value:

  1. In the **AWS IoT console**, choose **Manage** and then choose **Things**.
  2. Choose the thing you created for your sample app (for example, **My_light_bulb**) and then choose **Interact**.

  On the thing details page, your endpoint is displayed in the **HTTPS** section. You should also see a message that says: `This thing already appears to be connected.`

- **Check certificate activation**

  Certificates authenticate your device with AWS IoT Core.

  To check whether your certificate is active:

  1. In the **AWS IoT console**, choose **Manage** and then choose **Things**.
  2. Choose the thing you created for your sample app (for example, **My_light_bulb**) and then choose **Security**.
  3. Select the certificate and then, from the certificate's details page, choose Select the certificate and then, from the certificate's details page, choose **Actions**.

  If in the dropdown list **Activate** isn't available and you can only choose **Deactivate**, your certificate is active. If not, choose **Activate** and rerun the sample program.

  If the program still doesn't run, check the certificate file names in the `certs` folder.

- **Check the policy attached to the thing resource**

  While certificates authenticate your device, AWS IoT policies permit the device to perform AWS IoT operations, such as subscribing or publishing to MQTT reserved topics.

  To check whether the correct policy is attached:

  1. Find the certificate as described previously, and then choose **Policies**.
  2. Choose the policy displayed and check whether it describes the `connect`, `subscribe`, `receive`, and `publish` actions that give the device permission to publish and subscribe to the MQTT reserved topics.

For a sample policy, see [Step 1: Create an AWS IoT policy for the Device Shadow](#).

If you see error messages that indicate trouble connecting to AWS IoT, it could be because of the permissions you're using for the policy. If that's the case, we recommend that you start with a policy that provides full access to AWS IoT resources and then rerun the sample program. You can either edit the current policy, or choose the current policy, choose **Detach**, and then create another policy that provides full access and attach it to your thing resource. You can later restrict the policy to only the actions and policies you need to run the program.

```
{
  "Version":"2012-10-17",
  "Statement": [
      {
          "Effect": "Allow",
          "Action": [
              "iot:*"
          ],
          "Resource": "*"
      }
  ]
}
```

- **Check your Device SDK installation**

  If the program still doesn't run, you can reinstall the Device SDK to make sure that your SDK installation is complete and correct.

Step 4: Review the results and next steps

In this tutorial, you learned how to:

- Install the required software, tools, and the AWS IoT Device SDK for Python.
- Understand how the sample app, `shadow.py`, uses the MQTT protocol for retrieving and updating the shadow's current state.
- Run the sample app for Device Shadows and observe the update to the Shadow document in the AWS IoT console. You also learned to troubleshoot any issues and fix errors when running the program.

Next steps

You can now run the `shadow.py` sample application and use Device Shadows to control the state. You can observe the updates to the Shadow document in the AWS IoT Console and observe delta events that the sample app responds to. Using the MQTT test client, you can subscribe to the reserved shadow topics and observe messages received by the topics when running the sample program. For more information about how to run this tutorial, see Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client.

Tutorial: Interacting with Device Shadow using the sample app and the MQTT test client

To interact with the `shadow.py` sample app, enter a value in the terminal for the `desired` value. For example, you can specify colors that resemble the traffic lights and AWS IoT responds to the request and updates the reported values.

In this tutorial, you'll learn how to:

- Use the `shadow.py` sample app to specify desired states and update the shadow's current state.
- Edit the Shadow document to observe delta events and how the `shadow.py` sample app responds to it.
- Use the MQTT test client to subscribe to shadow topics and observe updates when you run the sample program.

Before you run this tutorial, you must have:

Set up your AWS account, configured your Raspberry Pi device, and created an AWS IoT thing and policy. You must have also installed the required software, Device SDK, certificate files, and run the sample program in the terminal. For more information, see the previous tutorials Tutorial: Preparing your Raspberry Pi to run the shadow application and Step 1: Run the shadow.py sample app. You must complete these tutorials if you haven't already.

In this tutorial, you'll:

- Step 1: Update desired and reported values using shadow.py sample app
- Step 2: View messages from the shadow.py sample app in the MQTT test client
- Step 3: Troubleshoot errors with Device Shadow interactions
- Step 4: Review the results and next steps

This tutorial takes about 45 minutes to complete.

Step 1: Update desired and reported values using `shadow.py` sample app

In the previous tutorial Step 1: Run the shadow.py sample app, you learned how to observe a message published to the Shadow document in the AWS IoT console when you enter a desired value as described in the section Tutorial: Installing the Device SDK and running the sample application for Device Shadows.

In the previous example, we set the desired color to `yellow`. After you enter each value, the terminal prompts you to enter another `desired` value. If you again enter the same value (`yellow`), the app recognizes this and prompts you to enter a new `desired` value.

```
Enter desired value:
yellow
Local value is already 'yellow'.
Enter desired value:
```

Now, say that you enter the color `green`. AWS IoT responds to the request and updates the `reported` value to `green`. This is how the update happens when the `desired` state is different from the `reported` state, causing a delta.

How the `shadow.py` sample app simulates Device Shadow interactions:

1. Enter a `desired` value (say `yellow`) in the terminal to publish the desired state.
2. As the `desired` state is different from the `reported` state (say the color `green`), a delta occurs, and the app that is subscribed to the delta receives this message.
3. The app responds to the message and updates its state to the `desired` value, `yellow`.
4. The app then publishes an update message with the new reported value of the device's state, `yellow`.

Following shows the messages displayed in the terminal that shows how the update request is published.

```
Enter desired value:
green
Changed local shadow value to 'green'.
Updating reported shadow value to 'green'...
```

```
Update request published.
Finished updating reported shadow value to 'green'.
```

In the AWS IoT console, the Shadow document reflects the updated value to `green` for both the `reported` and `desired` fields, and the version number is incremented by 1. For example, if the previous version number was displayed as 10, the current version number will display as 11.

Note

Deleting a shadow doesn't reset the version number to 0. You'll see that the shadow version is incremented by 1 when you publish an update request or create another shadow with the same name.

Edit the Shadow document to observe delta events

The `shadow.py` sample app is also subscribed to `delta` events, and responds when there is a change to the `desired` value. For example, you can change the `desired` value to the color `red`. To do this, in the AWS IoT console, edit the Shadow document by clicking **Edit** and then set the `desired` value to `red` in the JSON, while keeping the `reported` value to `green`. Before you save the changes, keep the terminal on the Raspberry Pi open as you'll see messages displayed in the terminal when the change occurs.

```
{
"desired": {
  "welcome": "aws-iot",
  "color": "red"
},
"reported": {
  "welcome": "aws-iot",
  "color": "green"
}
}
```

After you save the new value, the `shadow.py` sample app responds to this change and displays messages in the terminal indicating the delta. You should then see the following messages appear below the prompt for entering the `desired` value.

```
Enter desired value:
```

```
Received shadow delta event.
Delta reports that desired value is 'red'. Changing local value...
Changed local shadow value to 'red'.
Updating reported shadow value to 'red'...
Finished updating reported shadow value to 'red'.
Enter desired value:
Update request published.
Finished updating reported shadow value to 'red'.
```

Step 2: View messages from the `shadow.py` sample app in the MQTT test client

You can use the **MQTT test client** in the **AWS IoT console** to monitor MQTT messages that are passed in your AWS account. By subscribing to reserved MQTT topics used by the Device Shadow service, you can observe the messages received by the topics when running the sample app.

If you haven't already used the MQTT test client, you can review View MQTT messages with the AWS IoT MQTT client. This helps you learn how to use the **MQTT test client** in the **AWS IoT console** to view MQTT messages as they pass through the message broker.

1. **Open the MQTT test client**

   Open the MQTT test client in the AWS IoT console in a new window so that you can observe the messages received by the MQTT topics without losing the configuration of your MQTT test client. The MQTT test client doesn't retain any subscriptions or message logs if you leave it to go to another page in the console. For this section of the tutorial, you can have the Shadow document of your AWS IoT thing and the MQTT test client open in separate windows to more easily observe the interaction with Device Shadows.

2. **Subscribe to the MQTT reserved Shadow topics**

   You can use the MQTT test client to enter the names of the Device Shadow's MQTT reserved topics and subscribe to them to receive updates when running the `shadow.py` sample app. To subscribe to the topics:

   a. In the **MQTT test client** in the **AWS IoT console**, choose **Subscribe to a topic**.

   b. In the **Topic filter** section, enter: **$aws/things/*thingname*/shadow/update/#**. Here, `thingname` is the name of the thing resource that you created earlier (for example, `My_light_bulb`).

   c. Keep the default values for the additional configuration settings, and then choose **Subscribe**.

By using the **#** wildcard in the topic subscription, you can subscribe to multiple MQTT topics at the same time and observe all the messages that are exchanged between the device and its Shadow in a single window. For more information about the wildcard characters and their use, see [MQTT topics](#).

3. **Run `shadow.py` sample program and observe messages**

   In your command line window of the Raspberry Pi, if you've disconnected the program, run the sample app again and watch the messages in the **MQTT test client** in the **AWS IoT console**.

   a. Run the following command to restart the sample program. Replace *`your-iot-thing-name`* and *`your-iot-endpoint`* with the names of the AWS IoT thing that you created earlier (for example, `My_light_bulb`), and the endpoint to interact with the device.

   ```
   cd ~/aws-iot-device-sdk-python-v2/samples/service-clients
   python3 shadow.py --ca_file ~/certs/Amazon-root-CA-1.pem --cert ~/certs/
   device.pem.crt --key ~/certs/private.pem.key --endpoint your-iot-endpoint --
   thing_name your-iot-thing-name
   ```

   The `shadow.py` sample app then runs and retrieves the current shadow state. If you've deleted the shadow or cleared the current states, the program sets the current value to `off` and then prompts you to enter a `desired` value.

   ```
   Connecting to a3qEXAMPLEffp-ats.iot.us-west-2.amazonaws.com with client ID
    'test-0c8ae2ff-cc87-49d2-a82a-ae7ba1d0ca5a'...
   Connected!
   Subscribing to Delta events...
   Subscribing to Update responses...
   Subscribing to Get responses...
   Requesting current shadow state...
   Launching thread to read user input...
   Finished getting initial shadow state.
   Shadow document lacks 'color' property. Setting defaults...
   Changed local shadow value to 'off'.
   Updating reported shadow value to 'off'...
   Update request published.
   Finished updating reported shadow value to 'off'...
   Enter desired value:
   ```

On the other hand, if the program was running and you restarted it, you'll see the latest color value reported in the terminal. In the MQTT test client, you'll see an update to the topics **$aws/things/*thingname*/shadow/get** and **$aws/things/*thingname*/shadow/get/accepted**.

Suppose that the latest color reported was `green`. Following shows the contents of the **$aws/things/*thingname*/shadow/get/accepted** JSON file.

```
{
"state": {
  "desired": {
    "welcome": "aws-iot",
    "color": "green"
  },
  "reported": {
    "welcome": "aws-iot",
    "color": "green"
  }
},
"metadata": {
  "desired": {
    "welcome": {
      "timestamp": 1620156892
    },
    "color": {
      "timestamp": 1620161643
    }
  },
  "reported": {
    "welcome": {
      "timestamp": 1620156892
    },
    "color": {
      "timestamp": 1620161643
    }
  }
},
"version": 10,
"timestamp": 1620173908
}
```

b. Enter a `desired` value in the terminal, such as `yellow`. The `shadow.py` sample app responds and displays the following messages in the terminal that show the change in the `reported` value to `yellow`.

```
Enter desired value:
yellow
Changed local shadow value to 'yellow'.
Updating reported shadow value to 'yellow'...
Update request published.
Finished updating reported shadow value to 'yellow'.
```

In the **MQTT test client** in the **AWS IoT console**, under **Subscriptions**, you see that the following topics received a message:

- **$aws/things/*thingname*/shadow/update**: shows that both `desired` and `updated` values change to the color `yellow`.
- **$aws/things/*thingname*/shadow/update/accepted**: shows the current values of the `desired` and `reported` states and their metadata and version information.
- **$aws/things/*thingname*/shadow/update/documents**: shows the previous and current values of the `desired` and `reported` states and their metadata and version information.

As the document **$aws/things/*thingname*/shadow/update/documents** also contains information that is contained in the other two topics, we can review it to see the state information. The previous state shows the reported value set to `green`, its metadata and version information, and the current state that shows the reported value updated to `yellow`.

```
{
"previous": {
  "state": {
    "desired": {
      "welcome": "aws-iot",
      "color": "green"
    },
    "reported": {
      "welcome": "aws-iot",
      "color": "green"
```

```
      }
    },
    "metadata": {
      "desired": {
        "welcome": {
          "timestamp": 1617297888
        },
        "color": {
          "timestamp": 1617297898
        }
      },
      "reported": {
        "welcome": {
          "timestamp": 1617297888
        },
        "color": {
          "timestamp": 1617297898
        }
      }
    },
    "version": 10
  },
  "current": {
    "state": {
      "desired": {
        "welcome": "aws-iot",
        "color": "yellow"
      },
      "reported": {
        "welcome": "aws-iot",
        "color": "yellow"
      }
    },
    "metadata": {
      "desired": {
        "welcome": {
          "timestamp": 1617297888
        },
        "color": {
          "timestamp": 1617297904
        }
      },
      "reported": {
        "welcome": {
```

```
        "timestamp": 1617297888
      },
      "color": {
        "timestamp": 1617297904
      }
    }
  },
  "version": 11
},
"timestamp": 1617297904
}
```

c. Now, if you enter another `desired` value, you see further changes to the `reported` values and message updates received by these topics. The version number also increments by 1. For example, if you enter the value `green`, the previous state reports the value `yellow` and the current state reports the value `green`.

4. **Edit Shadow document to observe delta events**

   To observe changes to the delta topic, edit the Shadow document in the AWS IoT console. For example, you can change the `desired` value to the color `red`. To do this, in the AWS IoT console, choose **Edit** and then set the `desired` value to red in the JSON, while keeping the `reported` value set to `green`. Before you save the change, keep the terminal open as you'll see the delta message reported in the terminal.

```
{
"desired": {
  "welcome": "aws-iot",
  "color": "red"
},
"reported": {
  "welcome": "aws-iot",
  "color": "green"
}
}
```

   The `shadow.py` sample app responds to this change and displays messages in the terminal indicating the delta. In the MQTT test client, the `update` topics will have received a message showing changes to the `desired` and `reported` values.

   You also see that the topic **$aws/things/*thingname*/shadow/update/delta** received a message. To see the message, choose this topic, which is listed under **Subscriptions**.

```
{
"version": 13,
"timestamp": 1617318480,
"state": {
  "color": "red"
},
"metadata": {
  "color": {
    "timestamp": 1617318480
  }
}
}
```

Step 3: Troubleshoot errors with Device Shadow interactions

When you run the Shadow sample app, you might encounter issues with observing interactions with the Device Shadow service.

If the program runs successfully and prompts you to enter a `desired` value, you should be able to observe the Device Shadow interactions by using the Shadow document and the MQTT test client as described previously. However, if you're unable to see the interactions, here are some things you can check:

- **Check the thing name and its shadow in the AWS IoT console**

  If you don't see the messages in the Shadow document, review the command and make sure it matches the thing name in the **AWS IoT console**. You can also check whether you have a classic shadow by choosing your thing resource and then choosing **Shadows**. This tutorial focuses primarily on interactions with the classic shadow.

  You can also confirm that the device you used is connected to the internet. In the **AWS IoT console**, choose the thing you created earlier, and then choose **Interact**. On the thing details page, you should see a message here that says: `This thing already appears to be connected.`
- **Check the MQTT reserved topics you subscribed to**

  If you don't see the messages appear in the MQTT test client, check whether the topics you subscribed to are formatted correctly. MQTT Device Shadow topics have a format **$aws/things/*thingname*/shadow/** and might have `update`, `get`, or `delete` following it depending on actions you want to perform on the shadow. This tutorial uses the topic **$aws/things/*thingname*/shadow/#** so make sure you entered it correctly when subscribing to the topic in the **Topic filter** section of the test client.

As you enter the topic name, make sure that the *thingname* is the same as the name of the AWS IoT thing that you created earlier. You can also subscribe to additional MQTT topics to see if an update has been successfully performed. For example, you can subscribe to the topic **$aws/things/*thingname*/shadow/update/rejected** to receive a message whenever an update request failed so that you can debug connection issues. For more information about the reserved topics, see the section called "Shadow topics" and Device Shadow MQTT topics.

Step 4: Review the results and next steps

In this tutorial, you learned how to:

- Use the `shadow.py` sample app to specify desired states and update the shadow's current state.
- Edit the Shadow document to observe delta events and how the `shadow.py` sample app responds to it.
- Use the MQTT test client to subscribe to shadow topics and observe updates when you run the sample program.

Next steps

You can subscribe to additional MQTT reserved topics to observe updates to the shadow application. For example, if you only subscribe to the topic **$aws/things/*thingname*/shadow/update/accepted**, you'll see only the current state information when an update is successfully performed.

You can also subscribe to additional shadow topics to debug issues or learn more about the Device Shadow interactions and also debug any issues with the Device Shadow interactions. For more information, see the section called "Shadow topics" and Device Shadow MQTT topics.

You can also choose to extend your application by using named shadows or by using additional hardware connected with the Raspberry Pi for the LEDs and observe changes to their state using messages sent from the terminal.

For more information about the Device Shadow service and using the service in devices, apps, and services, see AWS IoT Device Shadow service, Using shadows in devices, and Using shadows in apps and services.

Tutorial: Creating a custom authorizer for AWS IoT Core

This tutorial demonstrates the steps to create, validate, and use Custom Authentication by using the AWS CLI. Optionally, using this tutorial, you can use Postman to send data to AWS IoT Core by using the HTTP Publish API.

This tutorial show you how to create a sample Lambda function that implements the authorization and authentication logic and a custom authorizer using the **create-authorizer** call with token signing enabled. The authorizer is then validated using the **test-invoke-authorizer**, and finally you can send data to AWS IoT Core by using the HTTP Publish API to a test MQTT topic. Sample request will specify the authorizer to invoke by using the `x-amz-customauthorizer-name` header and pass the token-key-name and `x-amz-customauthorizer-signature` in request headers.

What you'll learn in this tutorial:

- How to create a Lambda function to be a custom authorizer handler
- How to create a custom authorizer using the AWS CLI with token signing enabled
- How to test your custom authorizer using the **test-invoke-authorizer** command
- How to publish an MQTT topic by using Postman and validate the request with your custom authorizer

This tutorial takes about 60 minutes to complete.

In this tutorial, you'll:

- Step 1: Create a Lambda function for your custom authorizer
- Step 2: Create a public and private key pair for your custom authorizer
- Step 3: Create a custom authorizer resource and its authorization
- Step 4: Test the authorizer by calling test-invoke-authorizer
- Step 5: Test publishing MQTT message using Postman
- Step 6: View messages in MQTT test client
- Step 7: Review the results and next steps
- Step 8: Clean up

Before you start this tutorial, make sure that you have:

- **Set up AWS account**

  You'll need your AWS account and AWS IoT console to complete this tutorial.

  The account you use for this tutorial works best when it includes at least these AWS managed policies:

  - `IAMFullAccess`
  - `AWSIoTFullAccess`
  - `AWSLambda_FullAccess`

  > **Important**
  >
  > The IAM policies used in this tutorial are more permissive than you should follow in a production implementation. In a production environment, make sure that your account and resource policies grant only the necessary permissions.
  > When you create IAM policies for production, determine what access users and roles need, and then design the policies that allow them to perform only those tasks.
  > For more information, see Security best practices in IAM

- **Installed the AWS CLI**

  For information about how to install the AWS CLI, see Installing the AWS CLI. This tutorial requires AWS CLI version `aws-cli/2.1.3 Python/3.7.4 Darwin/18.7.0 exe/x86_64` or later.

- **OpenSSL tools**

  The examples in this tutorial use LibreSSL 2.6.5. You can also use OpenSSL v1.1.1i tools for this tutorial.

- **Reviewed the AWS Lambda overview**

  If you haven't used AWS Lambda before, review AWS Lambda and Getting started with Lambda to learn its terms and concepts.

- **Reviewed how to build requests in Postman**

  For more information, see Building requests.

- **Removed custom authorizers from previous tutorial**

Your AWS account can have only a limited number of custom authorizers configured at one time. For information about how to remove a custom authorizer, see the section called “Step 8: Clean up”.

Step 1: Create a Lambda function for your custom authorizer

Custom authentication in AWS IoT Core uses authorizer resources that you create to authenticate and authorize clients. The function you'll create in this section authenticates and authorizes clients as they connect to AWS IoT Core and access AWS IoT resources.

The Lambda function does the following:

- If a request comes from **test-invoke-authorizer**, it returns an IAM policy with a `Deny` action.
- If a request comes from Postman using HTTP and the `actionToken` parameter has a value of `allow`, it returns an IAM policy with an `Allow` action. Otherwise, it returns an IAM policy with a `Deny` action.

To create the Lambda function for your custom authorizer

1. In the Lambda console, open Functions.
2. Choose **Create function**.
3. Confirm **Author from scratch** is selected.
4. Under **Basic information**:
   a. In **Function name**, enter **`custom-auth-function`**.
   b. In **Runtime**, confirm **Node.js 18.x**
5. Choose **Create function**.

   Lambda creates a Node.js function and an execution role that grants the function permission to upload logs. The Lambda function assumes the execution role when you invoke your function and uses the execution role to create credentials for the AWS SDK and to read data from event sources.
6. To see the function's code and configuration in the AWS Cloud9 editor, choose **custom-auth-function** in the designer window, and then choose **index.js** in the navigation pane of the editor.

For scripting languages such as Node.js, Lambda includes a basic function that returns a success response. You can use the [AWS Cloud9](#) editor to edit your function as long as your source code doesn't exceed 3 MB.

7. Replace the **index.js** code in the editor with the following code:

```
// A simple Lambda function for an authorizer. It demonstrates
// How to parse a CLI and Http password to generate a response.

export const handler = async (event, context, callback) => {

    //Http parameter to initiate allow/deny request
    const HTTP_PARAM_NAME='actionToken';
    const ALLOW_ACTION = 'Allow';
    const DENY_ACTION = 'Deny';

    //Event data passed to Lambda function
    var event_str = JSON.stringify(event);
    console.log('Complete event :'+ event_str);

    //Read protocolData from the event json passed to Lambda function
    var protocolData = event.protocolData;
    console.log('protocolData value---> ' + protocolData);

    //Get the dynamic account ID from function's ARN to be used
    // as full resource for IAM policy
    var ACCOUNT_ID = context.invokedFunctionArn.split(":")[4];
    console.log("ACCOUNT_ID---"+ACCOUNT_ID);

    //Get the dynamic region from function's ARN to be used
    // as full resource for IAM policy
    var REGION = context.invokedFunctionArn.split(":")[3];
    console.log("REGION---"+REGION);

    //protocolData data will be undefined if testing is done via CLI.
    // This will help to test the set up.
    if (protocolData === undefined) {

        //If CLI testing, pass deny action as this is for testing purpose only.
        console.log('Using the test-invoke-authorizer cli for testing only');
        callback(null, generateAuthResponse(DENY_ACTION,ACCOUNT_ID,REGION));

    } else{
```

```
        //Http Testing from Postman
        //Get the query string from the request
        var queryString = event.protocolData.http.queryString;
        console.log('queryString values -- ' + queryString);
        /*         global URLSearchParams       */
        const params = new URLSearchParams(queryString);
        var action = params.get(HTTP_PARAM_NAME);

        if(action!=null && action.toLowerCase() === 'allow'){

            callback(null, generateAuthResponse(ALLOW_ACTION,ACCOUNT_ID,REGION));

        }else{

            callback(null, generateAuthResponse(DENY_ACTION,ACCOUNT_ID,REGION));

        }

    }

};

// Helper function to generate the authorization IAM response.
var generateAuthResponse = function(effect,ACCOUNT_ID,REGION) {

    var full_resource = "arn:aws:iot:"+ REGION + ":" + ACCOUNT_ID + ":*";
    console.log("full_resource---"+full_resource);

    var authResponse = {};
    authResponse.isAuthenticated = true;
    authResponse.principalId = 'principalId';

    var policyDocument = {};
    policyDocument.Version = '2012-10-17       ';
    policyDocument.Statement = [];
    var statement = {};
    statement.Action = 'iot:*';
    statement.Effect = effect;
    statement.Resource = full_resource;
    policyDocument.Statement[0] = statement;
    authResponse.policyDocuments = [policyDocument];
    authResponse.disconnectAfterInSeconds = 3600;
    authResponse.refreshAfterInSeconds = 600;
```

```
    console.log('custom auth policy function called from http');
    console.log('authResponse --> ' + JSON.stringify(authResponse));
    console.log(authResponse.policyDocuments[0]);

    return authResponse;
}
```

8. Choose **Deploy**.
9. After **Changes deployed** appears above the editor:
   a. Scroll to the **Function overview** section above the editor.
   b. Copy the **Function ARN** and save it to use later in this tutorial.
10. Test your function.
    a. Choose the **Test** tab.
    b. Using the default test settings, choose **Invoke**.
    c. If the test succeeded, in the **Execution results**, open the **Details** view. You should see the policy document that the function returned.

       If the test failed or you don't see a policy document, review the code to find and correct the errors.

Step 2: Create a public and private key pair for your custom authorizer

Your custom authorizer requires a public and private key to authenticate it. The commands in this section use OpenSSL tools to create this key pair.

To create the public and private key pair for your custom authorizer

1. Create the private key file.

   ```
   openssl genrsa -out private-key.pem 4096
   ```

2. Verify the private key file you just created.

   ```
   openssl rsa -check -in private-key.pem -noout
   ```

   If the command doesn't display any errors, the private key file is valid.

3. Create the public key file.

```
openssl rsa -in private-key.pem -pubout -out public-key.pem
```

4. Verify the public key file.

```
openssl pkey -inform PEM -pubin -in public-key.pem -noout
```

If the command doesn't display any errors, the public key file is valid.

Step 3: Create a custom authorizer resource and its authorization

The AWS IoT custom authorizer is the resource that ties together all the elements created in the previous steps. In this section, you'll create a custom authorizer resource and give it permission to run the Lambda function you created earlier. You can create a custom authorizer resource by using the AWS IoT console, the AWS CLI, or the AWS API.

For this tutorial, you only need to create one custom authorizer. This section describes how to create by using the AWS IoT console and the AWS CLI, so you can use the method that is most convenient for you. There's no difference between the custom authorizer resources created by either method.

Create a custom authorizer resource

Choose one of these options to create your custom authorizer resource

- Create a custom authorizer by using the AWS IoT console
- Create a custom authorizer using the AWS CLI

To create a custom authorizer (console)

1. Open the Custom authorizer page of the AWS IoT console, and choose **Create Authorizer**.
2. In **Create Authorizer**:

   a. In **Authorizer name**, enter **`my-new-authorizer`**.
   b. In **Authorizer status**, check **Active**.
   c. In **Authorizer function**, choose the Lambda function you created earlier.
   d. In **Token validation - optional**:

i. Toggle on **Token validation**.

ii. In **Token key name**, enter **`tokenKeyName`**.

iii. Choose **Add key**.

iv. In **Key name**, enter **`FirstKey`**.

v. In **Public key**, enter the contents of the `public-key.pem` file. Be sure to include the lines from the file with `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` and don't add or remove any line feeds, carriage returns, or other characters from the file contents. The string that you enter should look something like this example.

```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvEBzOk4vhN+3LgslvEWt
sLCqNmt5Damas3bmiTRvq2gjRJ6KXGTGQChqArAJwL1a9dkS9+maaXC3vc6xzx9z
QPu/vQOe5tyzz1MsKdmtFGxMqQ3qjEXAMPLEOmqyUKPP5mff58k6ePSfXAnzBH0q
lg2HioefrpU5OSAnpuRAjYKofKjbc2Vrn6N2G7hV+IfTBvCElf0csalS/Rk4phD5
oa4Y0GHISRnevypg5C8n9Rrz91PWGqP6M/q5DNJJXjMyleG92hQgu1N696bn5Dw8
FhedszFa6b2x6xrItZFzewNQkPMLMFhNrQIIyvshtT/F1LVCS5+v8AQ8UGGDfZmv
QeqAMAF7WgagDMXcfgKSVU8yid2sIm56qsCLMvD2Sq8Lgzpey9N5ON1o1Cvldwvc
KrJJtgwW6hVqRGuShnownLpgG86M6neZ5sRMbVNZO8OzcobLngJ0Ibw9KkcUdklW
gvZ6HEJqBY2XE70iEXAMPLETPHzhqvK6Ei1HGxpHsXx6BNft582J1VpgYjXha8oa
/NN7l7Zbj/euAb41IVtmX8JrD9z613d1iM5L8HluJlUzn62Q+VeNV2tdA7MfPfMC
8btGYladFAnitThaz6+F0VSBJPu7pZQoLnqyEp5zLMtF+kFl2yOBmGAP0RBivRd9
JWBUCG0bqcLQPeQyjbXSOfUCAwEAAQ==
-----END PUBLIC KEY-----
```

3. Choose **Create authorizer**.

4. If the custom authorizer resource was created, you'll see the list of custom authorizers and your new custom authorizer should appear in the list and you can continue to the next section to test it.

   If you see an error, review the error and try to create your custom authorizer again and double-check the entries. Note that each custom authorizer resource must have a unique name.

To create a custom authorizer (AWS CLI)

1. Substitute your values for `authorizer-function-arn` and `token-signing-public-keys`, and then run the following command:

```
aws iot create-authorizer \
--authorizer-name "my-new-authorizer" \
--token-key-name "tokenKeyName" \
--status ACTIVE \
--no-signing-disabled \
--authorizer-function-arn "arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function" \
--token-signing-public-keys FirstKey="-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvEBzOk4vhN+3LgslvEWt
sLCqNmt5Damas3bmiTRvq2gjRJ6KXGTGQChqArAJwL1a9dkS9+maaXC3vc6xzx9z
QPu/vQOe5tyzz1MsKdmtFGxMqQ3qjEXAMPLEOmqyUKPP5mff58k6ePSfXAnzBH0q
lg2HioefrpU5OSAnpuRAjYKofKjbc2Vrn6N2G7hV+IfTBvCElf0csalS/Rk4phD5
oa4Y0GHISRnevypg5C8n9Rrz91PWGqP6M/q5DNJJXjMyleG92hQgu1N696bn5Dw8
FhedszFa6b2x6xrItZFzewNQkPMLMFhNrQIIyvshtT/F1LVCS5+v8AQ8UGGDfZmv
QeqAMAF7WgagDMXcfgKSVU8yid2sIm56qsCLMvD2Sq8Lgzpey9N5ON1o1Cvldwvc
KrJJtgwW6hVqRGuShnownLpgG86M6neZ5sRMbVNZO8OzcobLngJ0Ibw9KkcUdklW
gvZ6HEJqBY2XE70iEXAMPLETPHzhqvK6Ei1HGxpHsXx6BNft582J1VpgYjXha8oa
/NN7l7Zbj/euAb41IVtmX8JrD9z613d1iM5L8HluJlUzn62Q+VeNV2tdA7MfPfMC
8btGYladFAnitThaz6+F0VSBJPu7pZQoLnqyEp5zLMtF+kFl2yOBmGAP0RBivRd9
JWBUCG0bqcLQPeQyjbXSOfUCAwEAAQ==
-----END PUBLIC KEY-----"
```

Where:

- The `authorizer-function-arn` value is the Amazon Resource Name (ARN) of the Lambda function you created for your custom authorizer.
- The `token-signing-public-keys` value includes the name of the key, **`FirstKey`**, and the contents of the `public-key.pem` file. Be sure to include the lines from the file with `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` and don't add or remove any line feeds, carriage returns, or other characters from the file contents.

  Note: be careful entering the public key as any alteration to the public key value makes it unusable.

2. If the custom authorizer is created, the command returns the name and ARN of the new resource, such as the following.

```
{
    "authorizerName": "my-new-authorizer",
    "authorizerArn": "arn:aws:iot:Region:57EXAMPLE833:authorizer/my-new-authorizer"
```

```
}
```

Save the `authorizerArn` value for use in the next step.

Remember that each custom authorizer resource must have a unique name.

Authorize the custom authorizer resource

In this section, you'll grant permission the custom authorizer resource that you just created permission to run the Lambda function. To grant the permission, you can use the [add-permission](https://docs.aws.amazon.com/cli/latest/reference/lambda/add-permission.html) CLI command.

Grant permission to your Lambda function using the AWS CLI

1. After inserting your values, enter the following command. Note that the `statement-id` value must be unique. Replace *`Id-1234`* with another value if you have run this tutorial before or if you get a `ResourceConflictException` error.

   ```
   aws lambda add-permission  \
   --function-name "custom-auth-function" \
   --principal "iot.amazonaws.com" \
   --action "lambda:InvokeFunction" \
   --statement-id "Id-1234" \
   --source-arn authorizerArn
   ```

2. If the command succeeds, it returns a permission statement, such as this example. You can continue to the next section to test the custom authorizer.

   ```
   {
       "Statement": "{\"Sid\":\"Id-1234\",\"Effect\":\"Allow\",\"Principal
   \":{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
   \",\"Resource\":\"arn:aws:lambda:Region:57EXAMPLE833:function:custom-
   auth-function\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
   \"arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function\"}}}"
   }
   ```

   If the command doesn't succeed, it returns an error, such as this example. You'll need to review and correct the error before you continue.

```
An error occurred (AccessDeniedException) when calling the AddPermission operation:
 User: arn:aws:iam::57EXAMPLE833:user/EXAMPLE-1 is not authorized to perform:
 lambda:AddPer
mission on resource: arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-
function
```

Step 4: Test the authorizer by calling test-invoke-authorizer

With all the resources defined, in this section, you'll call test-invoke-authorizer from the command line to test the authorization pass.

Note that when invoking the authorizer from the command line, `protocolData` is not defined, so the authorizer will always return a DENY document. This test does, however, confirm that your custom authorizer and Lambda function are configured correctly--even if it doesn't fully test the Lambda function.

To test your custom authorizer and its Lambda function by using the AWS CLI

1. In the directory that has the `private-key.pem` file you created in a previous step, run the following command.

   ```
   echo -n "tokenKeyValue" | openssl dgst -sha256 -sign private-key.pem | openssl
    base64 -A
   ```

   This command creates a signature string to use in the next step. The signature string looks something like this:


   ```
   dBwykzlb+fo+JmSGdwoGr8dyC2qB/IyLefJJr+rbCvmu9Jl4KHAA9DG+V
   +MMWu09YSA86+64Y3Gt4tOykpZqn9mn
   VB1wyxp+0bDZh8hmqUAUH3fwi3fPjBvCa4cwNuLQNqBZzbCvsluv7i2IMjEg
   +CPY0zrWt1jr9BikgGPDxWkjaeeh
   bQHHTo357TegKs9pP30Uf4TrxypNmFswA5k7QIc01n4bIyRTm90OyZ94R4bdJsHNig1JePgnuOBvMGCEFE09jGjj
   szEHfgAUAQIWXiVGQj16BU1xKpTGSiTAwheLKUjITOEXAMPLECK3aHKYKY
   +d1vTvdthKtYHBq8MjhzJ0kggbt29V
   QJCb8RilN/P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul/r/F3VV8ITtQp3aXiUtcspACi6ca
   +tsDuX
   f3LzCwQQF/YSUy02u5XkWn
   +sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230iqcXo3osjPha7JDyWM5o+K
   ```


```
EWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en1DAkc1a0s2U2UNm236EXAMPLELotyh7h
+flFeloZlAWQFH
xRlXsPqiVKS1ZIUClaZWprh/orDJplpiWfBgBIOgokJIDGP9gwhXIIk7zWrGmWpMK9o=
```

Copy this signature string to use in the next step. Be careful not to include any extra characters or leave any out.

2. In this command, replace the `token-signature` value with the signature string from the previous step and run this command to test your authorizer.

```
aws iot test-invoke-authorizer \
--authorizer-name my-new-authorizer \
--token tokenKeyValue \
--token-signature dBwykzlb+fo+JmSGdwoGr8dyC2qB/IyLefJJr
+rbCvmu9Jl4KHAA9DG+V+MMWu09YSA86+64Y3Gt4tOykpZqn9mnVB1wyxp
+0bDZh8hmqUAUH3fwi3fPjBvCa4cwNuLQNqBZzbCvsluv7i2IMjEg
+CPY0zrWt1jr9BikgGPDxWkjaeehbQHHTo357TegKs9pP30Uf4TrxypNmFswA5k7QIc01n4bIyRTm90OyZ94R4bdJsh
+d1vTvdthKtYHBq8MjhzJ0kggbt29VQJCb8RilN/
P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul/r/F3VV8ITtQp3aXiUtcspACi6ca
+tsDuXf3LzCwQQF/YSUy02u5XkWn
+sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230iqcXo3osjPha7JDyWM5o
+KEWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en1DAkc1a0s2U2UNm236EXAMPLELotyh7h
+flFeloZlAWQFHxRlXsPqiVKS1ZIUClaZWprh/orDJplpiWfBgBIOgokJIDGP9gwhXIIk7zWrGmWpMK9o=
```

If the command is successful, it returns the information generated by your custom authorizer function, such as this example.

```
{
    "isAuthenticated": true,
    "principalId": "principalId",
    "policyDocuments": [
        "{\"Version\":\"2012-10-17\",       \"Statement\":[{\"Action\":\"iot:*\",
\"Effect\":\"Deny\",\"Resource\":\"arn:aws:iot:Region:57EXAMPLE833:*\"}]}"
    ],
    "refreshAfterInSeconds": 600,
    "disconnectAfterInSeconds": 3600
}
```

If the command returns an error, review the error and double-check the commands you used in this section.

Step 5: Test publishing MQTT message using Postman

1. To get your device data endpoint from the command line, call [describe-endpoint](#) as shown here

```
aws iot describe-endpoint --output text --endpoint-type iot:Data-ATS
```

   Save this address for use as the *device_data_endpoint_address* in a later step.

2. Open a new Postman window and create a new HTTP POST request.

   a. From your computer, open the Postman app.

   b. In Postman, in the **File** menu, choose **New...**.

   c. In the **New** dialog box, choose **Request**.

   d. In Save request,

      i. In **Request name** enter **`Custom authorizer test request`**.

      ii. In **Select a collection or folder to save to:** choose or create a collection into which to save this request.

      iii. Choose **Save to *collection_name***.

3. Create the POST request to test your custom authorizer.

   a. In the request method selector next to the URL field, choose **POST**.

   b. In the URL field, create the URL for your request by using the following URL with the *device_data_endpoint_address* from the [describe-endpoint](#) command in a previous step.

```
https://device_data_endpoint_address:443/topics/test/cust-auth/topic?
qos=0&actionToken=allow
```

   Note that this URL includes the `actionToken=allow` query parameter that will tell your Lambda function to return a policy document that allows access to AWS IoT. After you enter the URL, the query parameters also appear in the **Params** tab of Postman.

   c. In the **Auth** tab, in the **Type** field, choose **No Auth**.

   d. In the Headers tab:

      i. If there's a **Host** key that's checked, uncheck this one.

ii. At the bottom of the list of headers add these new headers and confirm they are checked. Replace the **`Host`** value with your *`device_data_endpoint_address`* and the **`x-amz-customauthorizer-signature`** value with the signature string that you used with the **test-invoke-authorize** command in the previous section.

Key	Value
`x-amz-customauthorizer-name`	**`my-new-authorizer`**
`Host`	*`device_data_endpoint_address`*
`tokenKeyName`	**`tokenKeyValue`**

Key	Value
`x-amz-customauthorizer-signature`	*dBwykzlb+fo+JmSGdwoGr8dyC2qB/IyLefJJr+rbCvmu9Jl4KHAA9DG+V+MMWu09YSA86+64Y3Gt4tOykpZqn9mnVB1wyxp+0bDZh8hmqUAUH3fwi3fPjBvCa4cwNuLQNqBZzbCvsluv7i2IMjEg+CPY0zrWt1jr9BikgGPDxWkjaeehbQHHTo357TegKs9pP30Uf4TrxypNmFswA5k7QIc01n4bIyRTm90OyZ94R4bdJsHNig1JePgnuOBvMGCEFE09jGjjszEHfgAUAQIWXiVGQj16BU1xKpTGSiTAwheLKUjITOEXAMPLECK3aHKYKY+d1vTvdthKtYHBq8MjhzJ0kggbt29VQJCb8RilN/P5+vcVniSXWPplyB5jkYs9UvG08REoy64AtizfUhvSul/r/F3VV8ITtQp3aXiUtcspACi6ca+tsDuXf3LzCwQQF/YSUy02u5XkWn+sto6KCkpNlkD0wU8gl3+kOzxrthnQ8gEajd5Iylx230iqcXo3osjPha7JDyWM5o+KEWckTe91I1mokDr5sJ4JXixvnJTVSx1li49IalW4en1DAkc1a0s2U2UNm236EXAMPLELotyh7h+flFeloZlAWQFHxRlXsPqiVKS1ZIUClaZWprh/orDJplpiWfBgBIOgokJIDGP9gwhXIIk7zWrGmWpMK9o=*

e. In the Body tab:

   i. In the data format option box, choose **Raw**.

   ii. In the data type list, choose **JavaScript**.

   iii. In the text field, enter this JSON message payload for your test message:

```
{
    "data_mode": "test",
```

```
    "vibration": 200,
    "temperature": 40
}
```

4. Choose **Send** to send the request.

   If the request was successful, it returns:

   ```
   {
       "message": "OK",
       "traceId": "ff35c33f-409a-ea90-b06f-fbEXAMPLE25c"
   }
   ```

   The successful response indicates that your custom authorizer allowed the connection to AWS IoT and that the test message was delivered to broker in AWS IoT Core.

   If it returns an error, review error message, the *`device_data_endpoint_address`*, the signature string, and the other header values.

Keep this request in Postman for use in the next section.

Step 6: View messages in MQTT test client

In the previous step, you sent simulated device messages to AWS IoT by using Postman. The successful response indicated that your custom authorizer allowed the connection to AWS IoT and that the test message was delivered to broker in AWS IoT Core. In this section, you'll use the MQTT test client in the AWS IoT console to see the message contents from that message as other devices and services might.

To see the test messages authorized by your custom authorizer

1. In the AWS IoT console, open the MQTT test client.
2. In the **Subscribe to topic** tab, in **Topic filter**, enter **`test/cust-auth/topic`**, which is the message topic used in the Postman example from the previous section.
3. Choose **Subscribe**.

   Keep this window visible for the next step.
4. In Postman, in the request you created for the previous section, choose **Send**.

Review the response to make sure it was successful. If not, troubleshoot the error as the previous section describes.

5. In the **MQTT test client**, you should see a new entry that shows the message topic and, if expanded, the message payload from the request you sent from Postman.

   If you don't see your messages in the **MQTT test client**, here are some things to check:

   - Make sure your Postman request returned successfully. If AWS IoT rejects the connection and returns an error, the message in the request doesn't get passed to the message broker.
   - Make sure the AWS account and AWS Region used to open the AWS IoT console are the same as you're using in the Postman URL.
   - Make sure that you're using the appropriate endpoint for the custom authorizer. The default IoT endpoint might not support using custom authorizers with Lambda functions. Instead, you can use domain configurations to define a new endpoint and then specify that endpoint for the custom authorizer.
   - Make sure you've entered the topic correctly in the **MQTT test client**. The topic filter is case-sensitive. If in doubt, you can also subscribe to the **#** topic, which subscribes to all MQTT messages that pass through the message broker the AWS account and AWS Region used to open the AWS IoT console.

Step 7: Review the results and next steps

In this tutorial:

- You created a Lambda function to be a custom authorizer handler
- You created a custom authorizer with token signing enabled
- You tested your custom authorizer using the **test-invoke-authorizer** command
- You published an MQTT topic by using Postman and validate the request with your custom authorizer
- You used the **MQTT test client** to view the messages sent from your Postman test

Next steps

After you send some messages from Postman to verify that the custom authorizer is working, try experimenting to see how changing different aspects of this tutorial affect the results. Here are some examples to get you started.

- Change the signature string so that it's no longer valid to see how unauthorized connection attempts are handled. You should get an error response, such as this one, and the message should not appear in the **MQTT test client**.

```
{
    "message": "Forbidden",
    "traceId": "15969756-a4a4-917c-b47a-5433e25b1356"
}
```

- To learn more about how to find errors that might occur while you're developing and using AWS IoT rules, see Monitoring AWS IoT.

Step 8: Clean up

If you'd like repeat this tutorial, you might need to remove some of your custom authorizers. Your AWS account can have only a limited number of custom authorizers configured at one time and you can get a `LimitExceededException` when you try to add a new one without removing an existing custom authorizer.

To remove a custom authorizer (console)

1. Open the Custom authorizer page of the AWS IoT console, and in the list of custom authorizers, find the custom authorizer to remove.
2. Open the Custom authorizer details page and, from the **Actions** menu, choose **Edit**.
3. Uncheck the **Activate authorizer**, and then choose **Update**.

   You can't delete a custom authorizer while it's active.
4. From the Custom authorizer details page, open the **Actions** menu, and choose **Delete**.

To remove a custom authorizer (AWS CLI)

1. List the custom authorizers that you have installed and find the name of the custom authorizer you want to delete.

```
aws iot list-authorizers
```

2. Set the custom authorizer to `inactive` by running this command after replacing *Custom_Auth_Name* with the `authorizerName` of the custom authorizer to delete.

```
aws iot update-authorizer --status INACTIVE --authorizer-name Custom_Auth_Name
```

3. Delete the custom authorizer by running this command after replacing *Custom_Auth_Name* with the `authorizerName` of the custom authorizer to delete.

```
aws iot delete-authorizer --authorizer-name Custom_Auth_Name
```

Tutorial: Monitoring soil moisture with AWS IoT and Raspberry Pi

This tutorial shows you how to use a Raspberry Pi, a moisture sensor, and AWS IoT to monitor the soil moisture level for a house plant or garden. The Raspberry Pi runs code that reads the moisture level and temperature from the sensor and then sends the data to AWS IoT. You create a rule in AWS IoT that sends an email to an address subscribed to an Amazon SNS topic when the moisture level falls below a threshold.

> **Note**
>
> This tutorial might not be up to date. Some references might have been superseded since this topic was originally published.

Contents

Prerequisites

To complete this tutorial, you need:

- An AWS account.
- An IAM user with administrator permissions.
- A development computer running Windows, macOS, Linux, or Unix to access the AWS IoT console.
- A Raspberry Pi 3B or 4B running the latest Raspberry Pi OS. For installation instructions, see Install an operating system on the Raspberry Pi website.
- A monitor, keyboard, mouse, and Wi-Fi network or Ethernet connection for your Raspberry Pi.
- A Raspberry Pi-compatible moisture sensor. The sensor used in this tutorial is an Adafruit STEMMA I2C Capacitive Moisture Sensor with a JST 4-pin to female socket cable header.

Setting up AWS IoT

To complete this tutorial, you need to create the following resources. To connect a device to AWS IoT, you create an IoT thing, a device certificate, and an AWS IoT policy.

- An AWS IoT thing.

  A thing represents a physical device (in this case, your Rasberry Pi) and contains static metadata about the device.
- A device certificate.

  All devices must have a device certificate to connect to and authenticate with AWS IoT.
- An AWS IoT policy.

  Each device certificate has one or more AWS IoT policies associated with it. These policies determine which AWS IoT resources the device can access.
- An AWS IoT root CA certificate.

  Devices and other clients use an AWS IoT root CA certificate to authenticate the AWS IoT server with which they are communicating. For more information, see Server authentication.
- An AWS IoT rule.

A rule contains a query and one or more rule actions. The query extracts data from device messages to determine if the message data should be processed. The rule action specifies what to do if the data matches the query.

- An Amazon SNS topic and topic subscription.

  The rule listens for moisture data from your Raspberry Pi. If the value is below a threshold, it sends a message to the Amazon SNS topic. Amazon SNS sends that message to all email addresses subscribed to the topic.

Step 1: Create the AWS IoT policy

Create an AWS IoT policy that allows your Raspberry Pi to connect and send messages to AWS IoT.

1. In the AWS IoT console, if a **Get started** button appears, choose it. Otherwise, in the navigation pane, expand **Security**, and then choose **Policies**.
2. If a **You don't have any policies yet** dialog box appears, choose **Create a policy**. Otherwise, choose **Create**.
3. Enter a name for the AWS IoT policy (for example, **`MoistureSensorPolicy`**).
4. In the **Add statements** section, replace the existing policy with the following JSON. Replace *`region`* and *`account`* with your AWS Region and AWS account number.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/RaspberryPi"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/update",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/delete",
```

```
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/get"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/update/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/delete/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/get/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/update/rejected",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
RaspberryPi/shadow/delete/rejected"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
RaspberryPi/shadow/update/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
RaspberryPi/shadow/delete/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
RaspberryPi/shadow/get/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
RaspberryPi/shadow/update/rejected",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
RaspberryPi/shadow/delete/rejected"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DeleteThingShadow"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:thing/RaspberryPi"
```

```
        }
    ]
}
```

5. Choose **Create**.

Step 2: Create the AWS IoT thing, certificate, and private key

Create a thing in the AWS IoT registry to represent your Raspberry Pi.

1. In the [AWS IoT console](), in the navigation pane, choose **Manage**, and then choose **Things**.
2. If a **You don't have any things yet** dialog box is displayed, choose **Register a thing**. Otherwise, choose **Create**.
3. On the **Creating AWS IoT things** page, choose **Create a single thing**.
4. On the **Add your device to the device registry** page, enter a name for your IoT thing (for example, **`RaspberryPi`**), and then choose **Next**. You can't change the name of a thing after you create it. To change a thing's name, you must create a new thing, give it the new name, and then delete the old thing.
5. On the **Add a certificate for your thing** page, choose **Create certificate**.
6. Choose the **Download** links to download the certificate, private key, and root CA certificate.

> **Important**
>
> This is the only time you can download your certificate and private key.

7. To activate the certificate, choose **Activate**. The certificate must be active for a device to connect to AWS IoT.
8. Choose **Attach a policy**.
9. For **Add a policy for your thing**, choose **MoistureSensorPolicy**, and then choose **Register Thing**.

Step 3: Create an Amazon SNS topic and subscription

Create an Amazon SNS topic and subscription.

1. From the [AWS SNS console](), in the navigation pane, choose **Topics**, and then choose **Create topic**.

2. Choose type as **Standard** and enter a name for the topic (for example, **`MoistureSensorTopic`**).
3. Enter a display name for the topic (for example, **`Moisture Sensor Topic`**). This is the name displayed for your topic in the Amazon SNS console.
4. Choose **Create topic**.
5. In the Amazon SNS topic detail page, choose **Create subscription**.
6. For **Protocol**, choose **Email**.
7. For **Endpoint**, enter your email address.
8. Choose **Create subscription**.
9. Open your email client and look for a message with the subject **`MoistureSensorTopic`**. Open the email and click the **Confirm subscription** link.

> **Important**
>
> You won't receive any email alerts from this Amazon SNS topic until you confirm the subscription.

You should receive an email message with the text you typed.

Step 4: Create an AWS IoT rule to send an email

An AWS IoT rule defines a query and one or more actions to take when a message is received from a device. The AWS IoT rules engine listens for messages sent by devices and uses the data in the messages to determine if some action should be taken. For more information, see [Rules for AWS IoT](#).

In this tutorial, your Raspberry Pi publishes messages on `aws/things/RaspberryPi/shadow/update`. This is an internal MQTT topic used by devices and the Thing Shadow service. The Raspberry Pi publishes messages that have the following form:

```
{
    "reported": {
        "moisture" : moisture-reading,
        "temp" : temperature-reading
    }
}
```

You create a query that extracts the moisture and temperature data from the incoming message. You also create an Amazon SNS action that takes the data and sends it to Amazon SNS topic subscribers if the moisture reading is below a threshold value.

Create an Amazon SNS rule

1. In the AWS IoT console, choose **Message routing** and then choose **Rules**. If a **You don't have any rules yet** dialog box appears, choose **Create a rule**. Otherwise, choose **Create rule**.
2. In the **Rule properties** page, enter a **Rule name** such as **`MoistureSensorRule`**, and provide a short **Rule description** such as **`Sends an alert when soil moisture level readings are too low`**.
3. Choose **Next** and configure your SQL statement. Choose **SQL version** as **2016-03-23**, and enter the following AWS IoT SQL query statement:

```
SELECT * FROM '$aws/things/RaspberryPi/shadow/update/accepted' WHERE
 state.reported.moisture < 400
```

   This statement triggers the rule action when the `moisture` reading is less than `400`.

   > **Note**
   >
   > You might have to use a different value. After you have the code running on your Raspberry Pi, you can see the values that you get from your sensor by touching the sensor, placing it in water, or placing it in a planter.

4. Choose **Next** and attach rule actions. For **Action 1**, choose **Simple Notification Service**. The description for this rule action is **Send a message as an SNS push notification**.
5. For **SNS topic**, choose the topic that you created in Step 3: Create an Amazon SNS topic and subscription, **MoistureSensorTopic**, and leave the **Message format** as **RAW**. For **IAM role**, choose **Create a new role**. Enter a name for the role, for example, **`LowMoistureTopicRole`**, and then choose **Create role**.
6. Choose **Next** to review and then choose **Create** to create the rule.

Setting up your Raspberry Pi and moisture sensor

Insert your microSD card into the Raspberry Pi, connect your monitor, keyboard, mouse, and, if you're not using Wi-Fi, Ethernet cable. Do not connect the power cable yet.

Connect the JST jumper cable to the moisture sensor. The other side of the jumper has four wires:

- Green: I2C SCL
- White: I2C SDA
- Red: power (3.5 V)
- Black: ground

Hold the Raspberry Pi with the Ethernet jack on the right. In this orientation, there are two rows of GPIO pins at the top. Connect the wires from the moisture sensor to the bottom row of pins in the following order. Starting at the left-most pin, connect red (power), white (SDA), and green (SCL). Skip one pin, and then connect the black (ground) wire. For more information, see Python Computer Wiring.

Attach the power cable to the Raspberry Pi and plug the other end into a wall socket to turn it on.

Configure your Raspberry Pi

1. On **Welcome to Raspberry Pi**, choose **Next**.
2. Choose your country, language, timezone, and keyboard layout. Choose **Next**.
3. Enter a password for your Raspberry Pi, and then choose **Next**.
4. Choose your Wi-Fi network, and then choose **Next**. If you aren't using a Wi-Fi network, choose **Skip**.
5. Choose **Next** to check for software updates. When the updates are complete, choose **Restart** to restart your Raspberry Pi.

After your Raspberry Pi starts up, enable the I2C interface.

1. In the upper left corner of the Raspbian desktop, click the Raspberry icon, choose **Preferences**, and then choose **Raspberry Pi Configuration**.
2. On the **Interfaces** tab, for **I2C**, choose **Enable**.
3. Choose **OK**.

The libraries for the Adafruit STEMMA moisture sensor are written for CircuitPython. To run them on a Raspberry Pi, you need to install the latest version of Python 3.

1. Run the following commands from a command prompt to update your Raspberry Pi software:

   ```
   sudo apt-get update
   ```

   ```
   sudo apt-get upgrade
   ```
2. Run the following command to update your Python 3 installation:

   ```
   sudo pip3 install --upgrade setuptools
   ```
3. Run the following command to install the Raspberry Pi GPIO libraries:

   ```
   pip3 install RPI.GPIO
   ```
4. Run the following command to install the Adafruit Blinka libraries:

   ```
   pip3 install adafruit-blinka
   ```

   For more information, see [Installing CircuitPython Libraries on Raspberry Pi](https://learn.adafruit.com/circuitpython-on-raspberrypi-linux/installing-circuitpython-on-raspberry-pi).
5. Run the following command to install the Adafruit Seesaw libraries:

   ```
   sudo pip3 install adafruit-circuitpython-seesaw
   ```
6. Run the following command to install the AWS IoT Device SDK for Python:

   ```
   pip3 install AWSIoTPythonSDK
   ```

Your Raspberry Pi now has all of the required libraries. Create a file called **`moistureSensor.py`** and copy the following Python code into the file:

```
from adafruit_seesaw.seesaw import Seesaw
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
from board import SCL, SDA

import logging
import time
import json
import argparse
import busio
```

```
# Shadow JSON schema:
#
# {
#   "state": {
#       "desired":{
#           "moisture":<INT VALUE>,
#           "temp":<INT VALUE>
#       }
#   }
# }

# Function called when a shadow is updated
def customShadowCallback_Update(payload, responseStatus, token):

    # Display status and data from update request
    if responseStatus == "timeout":
        print("Update request " + token + " time out!")

    if responseStatus == "accepted":
        payloadDict = json.loads(payload)
        print("~~~~~~~~~~~~~~~~~~~~~~~")
        print("Update request with token: " + token + " accepted!")
        print("moisture: " + str(payloadDict["state"]["reported"]["moisture"]))
        print("temperature: " + str(payloadDict["state"]["reported"]["temp"]))
        print("~~~~~~~~~~~~~~~~~~~~~~~\n\n")

    if responseStatus == "rejected":
        print("Update request " + token + " rejected!")

# Function called when a shadow is deleted
def customShadowCallback_Delete(payload, responseStatus, token):

     # Display status and data from delete request
    if responseStatus == "timeout":
        print("Delete request " + token + " time out!")

    if responseStatus == "accepted":
        print("~~~~~~~~~~~~~~~~~~~~~~~")
        print("Delete request with token: " + token + " accepted!")
        print("~~~~~~~~~~~~~~~~~~~~~~~\n\n")

    if responseStatus == "rejected":
        print("Delete request " + token + " rejected!")
```

```
# Read in command-line parameters
def parseArgs():

    parser = argparse.ArgumentParser()
    parser.add_argument("-e", "--endpoint", action="store", required=True, dest="host",
 help="Your device data endpoint")
    parser.add_argument("-r", "--rootCA", action="store", required=True,
 dest="rootCAPath", help="Root CA file path")
    parser.add_argument("-c", "--cert", action="store", dest="certificatePath",
 help="Certificate file path")
    parser.add_argument("-k", "--key", action="store", dest="privateKeyPath",
 help="Private key file path")
    parser.add_argument("-p", "--port", action="store", dest="port", type=int,
 help="Port number override")
    parser.add_argument("-n", "--thingName", action="store", dest="thingName",
 default="Bot", help="Targeted thing name")
    parser.add_argument("-id", "--clientId", action="store", dest="clientId",
 default="basicShadowUpdater", help="Targeted client id")

    args = parser.parse_args()
    return args


# Configure logging
# AWSIoTMQTTShadowClient writes data to the log
def configureLogging():

    logger = logging.getLogger("AWSIoTPythonSDK.core")
    logger.setLevel(logging.DEBUG)
    streamHandler = logging.StreamHandler()
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
 %(message)s')
    streamHandler.setFormatter(formatter)
    logger.addHandler(streamHandler)


# Parse command line arguments
args = parseArgs()

if not args.certificatePath or not args.privateKeyPath:
    parser.error("Missing credentials for authentication.")
    exit(2)
```

```
# If no --port argument is passed, default to 8883
if not args.port:
    args.port = 8883


# Init AWSIoTMQTTShadowClient
myAWSIoTMQTTShadowClient = None
myAWSIoTMQTTShadowClient = AWSIoTMQTTShadowClient(args.clientId)
myAWSIoTMQTTShadowClient.configureEndpoint(args.host, args.port)
myAWSIoTMQTTShadowClient.configureCredentials(args.rootCAPath, args.privateKeyPath,
 args.certificatePath)

# AWSIoTMQTTShadowClient connection configuration
myAWSIoTMQTTShadowClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTShadowClient.configureConnectDisconnectTimeout(10) # 10 sec
myAWSIoTMQTTShadowClient.configureMQTTOperationTimeout(5) # 5 sec

# Initialize Raspberry Pi's I2C interface
i2c_bus = busio.I2C(SCL, SDA)

# Intialize SeeSaw, Adafruit's Circuit Python library
ss = Seesaw(i2c_bus, addr=0x36)

# Connect to AWS IoT
myAWSIoTMQTTShadowClient.connect()

# Create a device shadow handler, use this to update and delete shadow document
deviceShadowHandler =
 myAWSIoTMQTTShadowClient.createShadowHandlerWithName(args.thingName, True)

# Delete current shadow JSON doc
deviceShadowHandler.shadowDelete(customShadowCallback_Delete, 5)

# Read data from moisture sensor and update shadow
while True:

    # read moisture level through capacitive touch pad
    moistureLevel = ss.moisture_read()

    # read temperature from the temperature sensor
    temp = ss.get_temp()

    # Display moisture and temp readings
    print("Moisture Level: {}".format(moistureLevel))
```

```
    print("Temperature: {}".format(temp))

    # Create message payload
    payload = {"state":{"reported":{"moisture":str(moistureLevel),"temp":str(temp)}}}

    # Update shadow
    deviceShadowHandler.shadowUpdate(json.dumps(payload), customShadowCallback_Update, 5)
    time.sleep(1)
```

Save the file to a place you can find it. Run `moistureSensor.py` from the command line with the following parameters:

endpoint

Your custom AWS IoT endpoint. For more information, see Device Shadow REST API.

rootCA

The full path to your AWS IoT root CA certificate.

cert

The full path to your AWS IoT device certificate.

key

The full path to your AWS IoT device certificate private key.

thingName

Your thing name (in this case, `RaspberryPi`).

clientId

The MQTT client ID. Use `RaspberryPi`.

The command line should look like this:

```
python3 moistureSensor.py --endpoint your-endpoint --rootCA ~/certs/AmazonRootCA1.pem --cert ~/certs/raspberrypi-certificate.pem.crt --key ~/certs/raspberrypi-private.pem.key --thingName RaspberryPi --clientId RaspberryPi
```

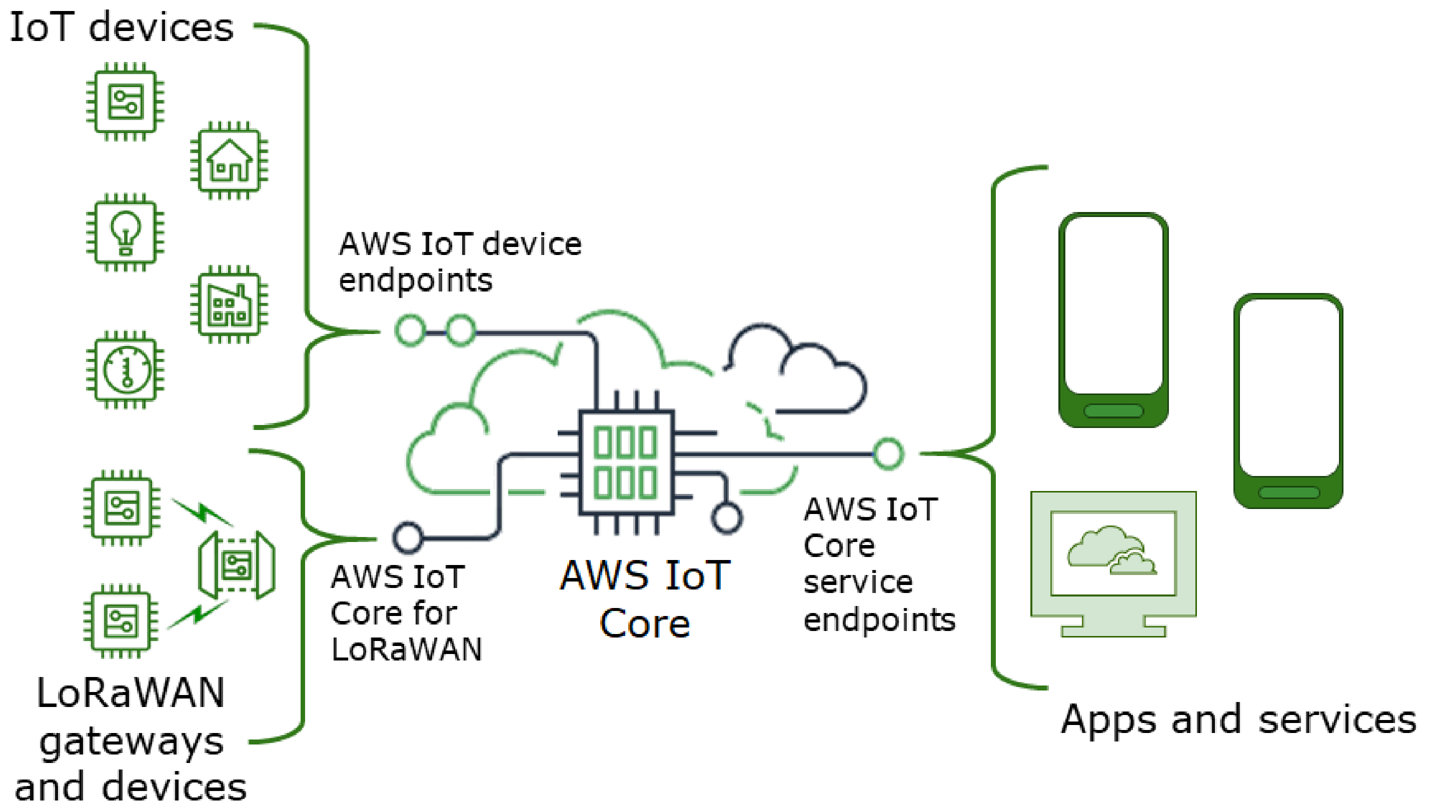
Try touching the sensor, putting it in a planter, or putting it in a glass of water to see how the sensor responds to various levels of moisture. If needed, you can change the threshold value in the

`MoistureSensorRule`. When the moisture sensor reading goes below the value specified in your rule's SQL query statement, AWS IoT publishes a message to the Amazon SNS topic. You should receive an email message that contains the moisture and temperature data.

After you have verified receipt of email messages from Amazon SNS, press **CTRL+C** to stop the Python program. It is unlikely that the Python program will send enough messages to incur charges, but it is a best practice to stop the program when you are done.

Connect to AWS IoT Core

AWS IoT Core supports connections with IoT devices, wireless gateways, services, and apps. Devices connect to AWS IoT Core so they can send data to and receive data from AWS IoT services and other devices. Apps and other services also connect to AWS IoT Core to control and manage the IoT devices and process the data from your IoT solution. This section describes how to choose the best way to connect and communicate with AWS IoT Core for each aspect of your IoT solution.



There are several ways to interact with AWS IoT. Apps and services can use the [AWS IoT Core - control plane endpoints] and devices can connect to AWS IoT Core by using the [AWS IoT device endpoints] or [AWS IoT Core for LoRaWAN Regions and endpoints].

AWS IoT Core - control plane endpoints

The **AWS IoT Core - control plane** endpoints provide access to functions that control and manage your AWS IoT solution.

- **Endpoints**

The **AWS IoT Core - control plane** and **AWS IoT Core Device Advisor control plane** endpoints are Region specific and are listed in AWS IoT Core Endpoints and Quotas. The formats of the endpoints are as follows.

Endpoint purpose	Endpoint format	Serves
AWS IoT Core - control plane	AWS IoT Control Plane endpoints	AWS IoT Control Plane API
AWS IoT Core Device Advisor - control plane	`api.iotdeviceadvisor.`*`aws-region`*`.amazonaws.com`	AWS IoT Core Device Advisor Control Plane API

- **IPv4 endpoints** — IPv4 endpoints support only IPv4 traffic, and are available for all Regions.

  IPv4 endpoints use the following naming convention:

  ```
  iot.aws-region.amazonaws.com
  ```

  For example the IPv4 endpoint name for the us-east-1 Region is `iot.us-east-1.amazonaws.com`.
- **Dual-stack (IPv4 and IPv6) endpoints** — Dual-stack endpoints support both IPv4 and IPv6 traffic. When a request is made to a dual-stack endpoint, the endpoint URL resolves to an IPv6 or an IPv4 address, depending on the protocol used by the network and client.

  Dual-stack endpoints use the following naming convention:

  ```
  iot.aws-region.api.aws
  ```

  For example the dual-stack endpoint name for the us-east-1 Region is `iot.us-east-1.api.aws`.
- **SDKs and tools**

The AWS SDKs provide language-specific support for the AWS IoT Core APIs, and the APIs of other AWS services. The AWS Mobile SDKs provide app developers with platform-specific support for the AWS IoT Core API, and other AWS services on mobile devices.

The AWS CLI provides command-line access to the functions provided by the AWS IoT service endpoints. AWS Tools for PowerShell provides tools to manage AWS services and resources in the PowerShell scripting environment.

- **Authentication**

  The service endpoints use IAM users and AWS credentials to authenticate users.

- **Learn more**

  For more information and links to SDK references, see the section called "Connect to AWS IoT Core service endpoints".

AWS IoT device endpoints

The AWS IoT device endpoints support communication between your IoT devices and AWS IoT.

- **Endpoints**

  The device endpoints support AWS IoT Core and AWS IoT Device Management functions. They are specific to your AWS account and you can see what they are by using the **describe-endpoint** command.

Endpoint purpose	Endpoint format	Serves
AWS IoT Core - data plane	See ???.	AWS IoT Data Plane API
AWS IoT Device Managemen t - jobs data	See ???.	AWS IoT Jobs Data Plane API
AWS IoT Device Advisor - data plane	See ???.	Not applicable
AWS IoT Device Managemen t - secure tunneling	`api.tunneling.iot.` *`aws-region`* `.amazonaws.com`	AWS IoT Secure Tunneling API

If you are using dual-stack endpoints (IPv4 and IPv6) for data plane operations, use the `iot:Data-ATS` endpoint type. `iot:Jobs` can be used for IPv4 only. For more information about these endpoints and the functions that they support, see the section called “AWS IoT device data and service endpoints”.

- **SDKs**

  The AWS IoT Device SDKs provide language-specific support for the Message Queueing Telemetry Transport (MQTT) and WebSocket Secure (WSS) protocols, which devices use to communicate with AWS IoT. AWS Mobile SDKs also provide support for MQTT device communications, AWS IoT APIs, and the APIs of other AWS services on mobile devices.

- **Authentication**

  The device endpoints use X.509 certificates or AWS IAM users with credentials to authenticate users.

- **Learn more**

  For more information and links to SDK references, see the section called “AWS IoT Device SDKs”.

AWS IoT Core for LoRaWAN gateways and devices

AWS IoT Core for LoRaWAN connects wireless gateways and devices to AWS IoT Core.

- **Endpoints**

  AWS IoT Core for LoRaWAN manages the gateway connections to account and Region-specific AWS IoT Core endpoints. Gateways can connect to your account's Configuration and Update Server (CUPS) endpoint that AWS IoT Core for LoRaWAN provides.

Endpoint purpose	Endpoint format	Serves
Configuration and Update Server (CUPS)	`account-specific-prefix .cups.lorawan. aws-region .amazonaws.com:443`	Gateway communication with the Configuration and Update Server provided by AWS IoT Core for LoRaWAN

Endpoint purpose	Endpoint format	Serves
LoRaWAN Network Server (LNS)	*account-specific-prefix* .gateway.lorawan. *aws-region* .amazonaws.com:443	Gateway communication with the LoRaWAN Network Server provided by AWS IoT Core for LoRaWAN

- **SDKs**

  The AWS IoT Wireless API that AWS IoT Core for LoRaWAN is built on is supported by the AWS SDK. For more information, see AWS SDKs and Toolkits.

- **Authentication**

  AWS IoT Core for LoRaWAN device communications use X.509 certificates to secure communications with AWS IoT.

- **Learn more**

  For more information about configuring and connecting wireless devices, see AWS IoT Core for LoRaWAN Regions and endpoints.

Connect to AWS IoT Core service endpoints

You can access the features of the **AWS IoT Core - control plane** by using the AWS CLI, the AWS SDK for your preferred language, or by calling the REST API directly. We recommend using the AWS CLI or an AWS SDK to interact with AWS IoT Core because they incorporate the best practices for calling AWS services. Calling the REST APIs directly is an option, but you must provide the necessary security credentials that enable access to the API.

> **Note**
>
> IoT devices should use AWS IoT Device SDKs. The Device SDKs are optimized for use on devices, support MQTT communication with AWS IoT, and support the AWS IoT APIs most used by devices. For more information about the Device SDKs and the features they provide, see AWS IoT Device SDKs.
> Mobile devices should use AWS Mobile SDKs. The Mobile SDKs provide support for AWS IoT APIs, MQTT device communications, and the APIs of other AWS services on mobile

devices. For more information about the Mobile SDKs and the features they provide, see AWS Mobile SDKs.

You can use AWS Amplify tools and resources in web and mobile applications to connect more easily to AWS IoT Core. For more information about connecting to AWS IoT Core by using Amplify, see PubSub in the Amplify documentation.

The following sections describe the tools and SDKs that you can use to develop and interact with AWS IoT and other AWS services. For the complete list of AWS tools and development kits that are available to build and manage apps on AWS, see Tools to Build on AWS.

AWS CLI for AWS IoT Core

The AWS CLI provides command-line access to AWS APIs.

- **Installation**

  For information about how to install the AWS CLI, see Installing the AWS CLI.
- **Authentication**

  The AWS CLI uses credentials from your AWS account.
- **Reference**

  For information about the AWS CLI commands for these AWS IoT Core services, see:
  - AWS CLI Command Reference for IoT
  - AWS CLI Command Reference for IoT data
  - AWS CLI Command Reference for IoT jobs data
  - AWS CLI Command Reference for IoT secure tunneling

For tools to manage AWS services and resources in the PowerShell scripting environment, see AWS Tools for PowerShell.

AWS SDKs

With AWS SDKs, your apps and compatible devices can call AWS IoT APIs and the APIs of other AWS services. This section provides links to the AWS SDKs and to the API reference documentation for the APIs of the AWS IoT Core services.

The AWS SDKs support these AWS IoT Core APIs

- AWS IoT
- AWS IoT Data Plane
- AWS IoT Jobs Data Plane
- AWS IoT Secure Tunneling
- AWS IoT Wireless

C++

To install the AWS SDK for C++ and use it to connect to AWS IoT:

1. Follow the instructions in Getting Started Using the AWS SDK for C++

   These instructions describe how to:

   - Install and build the SDK from source files
   - Provide credentials to use the SDK with your AWS account
   - Initialize and shutdown the SDK in your app or service
   - Create a CMake project to build your app or service
2. Create and run a sample app. For sample apps that use the AWS SDK for C++, see AWS SDK for C++ Code Examples.

Documentation for the AWS IoT Core services that the AWS SDK for C++ supports

- AWS::IoTClient" reference documentation
- Aws::IoTDataPlane::IoTDataPlaneClient reference documentation
- Aws::IoTJobsDataPlane::IoTJobsDataPlaneClient reference documentation
- Aws::IoTSecureTunneling::IoTSecureTunnelingClient reference documentation

Go

To install the AWS SDK for Go and use it to connect to AWS IoT:

1. Follow the instructions in Getting Started with the AWS SDK for Go

   These instructions describe how to:

- Install the AWS SDK for Go
- Get access keys for the SDK to access your AWS account
- Import packages into the source code of our apps or services

2. Create and run a sample app. For sample apps that use the AWS SDK for Go, see AWS SDK for Go Code Examples.

Documentation for the AWS IoT Core services that the AWS SDK for Go supports

- IoT reference documentation
- IoTDataPlane reference documentation
- IoTJobsDataPlane reference documentation
- IoTSecureTunneling reference documentation

Java

To install the AWS SDK for Java and use it to connect to AWS IoT:

1. Follow the instructions in Getting Started with AWS SDK for Java 2.x

   These instructions describe how to:

   - Sign up for AWS and Create an IAM User
   - Download the SDK
   - Set up AWS Credentials and Region
   - Use the SDK with Apache Maven
   - Use the SDK with Gradle
2. Create and run a sample app using one of the AWS SDK for Java 2.x Code Examples.
3. Review the SDK API reference documentation

Documentation for the AWS IoT Core services that the AWS SDK for Java supports

- IotClient reference documentation
- IotDataPlaneClient reference documentation
- IotJobsDataPlaneClient reference documentation
- IoTSecureTunnelingClient reference documentation

JavaScript

To install the AWS SDK for JavaScript and use it to connect to AWS IoT:

1. Follow the instructions in Setting Up the AWS SDK for JavaScript. These instructions apply to using the AWS SDK for JavaScript in the browser and with Node.JS. Make sure you follow the directions that apply to your installation.

   These instructions describe how to:

   - Check for the prerequisites
   - Install the SDK for JavaScript
   - Load the SDK for JavaScript
2. Create and run a sample app to get started with the SDK as the getting started option for your environment describes.
   - Get started with the AWS SDK for JavaScript in the Browser, or
   - Get started with the AWS SDK for JavaScript in Node.js

Documentation for the AWS IoT Core services that the AWS SDK for JavaScript supports

- `AWS.Iot reference documentation`
- `AWS.IotData reference documentation`
- `AWS.IotJobsDataPlane reference documentation`
- `AWS.IotSecureTunneling reference documentation`

.NET

To install the AWS SDK for .NET and use it to connect to AWS IoT:

1. Follow the instructions in Setting up your AWS SDK for .NET environment
2. Follow the instructions in Setting up your AWS SDK for .NET project

   These instructions describe how to:

   - Start a new project
   - Obtain and configure AWS credentials
   - Install AWS SDK packages

3. Create and run one of the sample programs in Working with AWS services in the AWS SDK for .NET
4. Review the SDK API reference documentation

Documentation for the AWS IoT Core services that the AWS SDK for .NET supports

- Amazon.IoT.Model reference documentation
- Amazon.IotData.Model reference documentation
- Amazon.IoTJobsDataPlane.Model reference documentation
- Amazon.IoTSecureTunneling.Model reference documentation

PHP

To install the AWS SDK for PHP and use it to connect to AWS IoT:

1. Follow the instructions in Getting Started with the AWS SDK for PHP Version 3

   These instructions describe how to:

   - Check for the prerequisites
   - Install the SDK
   - Apply the SDK to a PHP script
2. Create and run a sample app using one of the AWS SDK for PHP Version 3 Code Examples

Documentation for the AWS IoT Core services that the AWS SDK for PHP supports

- IoTClient reference documentation
- IoTDataPlaneClient reference documentation
- IoTJobsDataPlaneClient reference documentation
- IoTSecureTunnelingClient reference documentation

Python

To install the AWS SDK for Python (Boto3) and use it to connect to AWS IoT:

1. Follow the instructions in the AWS SDK for Python (Boto3) Quickstart

These instructions describe how to:

- Install the SDK
- Configure the SDK
- Use the SDK in your code

2. Create and run a sample program that uses the AWS SDK for Python (Boto3)

   This program displays the account's currently configured logging options. After you install the SDK and configure it for your account, you should be able to run this program.

```
import boto3
import json

# initialize client
iot = boto3.client('iot')

# get current logging levels, format them as JSON, and write them to stdout
response = iot.get_v2_logging_options()
print(json.dumps(response, indent=4))
```

   For more information about the function used in this example, see the section called "Configure AWS IoT logging".

Documentation for the AWS IoT Core services that the AWS SDK for Python (Boto3) supports

- IoT reference documentation
- IoTDataPlane reference documentation
- IoTJobsDataPlane reference documentation
- IoTSecureTunneling reference documentation

Ruby

To install the AWS SDK for Ruby and use it to connect to AWS IoT:

- Follow the instructions in Getting Started with the AWS SDK for Ruby

  These instructions describe how to:

  - Install the SDK

  - Configure the SDK
- Create and run the Hello World Tutorial

Documentation for the AWS IoT Core services that the AWS SDK for Ruby supports

- Aws::IoT::Client reference documentation
- Aws::IoTDataPlane::Client reference documentation
- Aws::IoTJobsDataPlane::Client reference documentation
- Aws::IoTSecureTunneling::Client reference documentation

AWS Mobile SDKs

The AWS Mobile SDKs provide mobile app developers platform-specific support for the APIs of the AWS IoT Core services, IoT device communication using MQTT, and the APIs of other AWS services.

Android

AWS Mobile SDK for Android

The AWS Mobile SDK for Android contains a library, samples, and documentation for developers to build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- AWS Mobile SDK for Android on GitHub
- AWS Mobile SDK for Android Readme
- AWS Mobile SDK for Android Samples
- AWS SDK for Android API reference
- AWSIoTClient Class reference documentation

iOS

AWS Mobile SDK for iOS

The AWS Mobile SDK for iOS is an open-source software development kit, distributed under an Apache Open Source license. The SDK for iOS provides a library, code samples, and documentation to help developers build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- AWS Mobile SDK for iOS on GitHub
- AWS SDK for iOS Readme
- AWS SDK for iOS Samples
- AWS IoT Class reference docs in the AWS SDK for iOS

REST APIs of the AWS IoT Core services

The REST APIs of the AWS IoT Core services can be called directly by using HTTP requests.

- **Endpoint URL**

  The service endpoints that expose the REST APIs of the AWS IoT Core services vary by Region and are listed in AWS IoT Core Endpoints and Quotas. You must use the endpoint for the Region that has the AWS IoT resources that you want to access, because AWS IoT resources are Region specific.
- **Authentication**

  The REST APIs of the AWS IoT Core services use AWS IAM credentials for authentication. For more information, see Signing AWS API requests in the AWS General Reference.
- **API reference**

  For information about the specific functions provided by the REST APIs of the AWS IoT Core services, see:
  - API reference for IoT.
  - API reference for IoT data.
  - API reference for IoT jobs data.
  - API reference for IoT secure tunneling.

Connect devices to AWS IoT

Devices connect to AWS IoT and other services through AWS IoT Core. Through AWS IoT Core, devices send and receive messages using device endpoints that are specific to your account. The

the section called "AWS IoT Device SDKs" support device communications using the MQTT and WSS protocols. For more information about the protocols that devices can use, see the section called "Device communication protocols".

The message broker

AWS IoT manages device communication through a message broker. Devices and clients publish messages to the message broker and also subscribe to messages that the message broker publishes. Messages are identified by an application-defined *topic*. When the message broker receives a message published by a device or client, it republishes that message to the devices and clients that have subscribed to the message's topic. The message broker also forwards messages to the AWS IoT rules engine, which can act on the content of the message.

AWS IoT message security

Device connections to AWS IoT use the section called "X.509 client certificates" and AWS signature V4 for authentication. Device communications are secured by TLS version 1.3 and AWS IoT requires devices to send the Server Name Indication (SNI) extension when they connect. For more information, see Transport Security in AWS IoT.

AWS IoT device data and service endpoints

> **Important**
>
> You can cache or store the endpoints in your device. This means you won't need to query the `DescribeEndpoint` API every time when a new device is connected. The endpoints won't change after AWS IoT Core creates them for your account.

Each account has several device endpoints that are unique to the account and support specific IoT functions. The AWS IoT device data endpoints support a publish/subscribe protocol that is designed for the communication needs of IoT devices; however, other clients, such as apps and services, can also use this interface if their application requires the specialized features that these endpoints provide. The AWS IoT device service endpoints support device-centric access to security and management services.

To learn your account's device data endpoint, you can find it in the **Settings** page of your AWS IoT Core console.

To learn your account's device endpoint for a specific purpose, including the device data endpoint, use the **describe-endpoint** CLI command shown here, or the `DescribeEndpoint` REST API, and provide the *`endpointType`* parameter value from the following table.

```
aws iot describe-endpoint --endpoint-type endpointType
```

This command returns an *`iot-endpoint`* in the following format: *`account-specific-prefix`*`.iot.`*`aws-region`*`.amazonaws.com`.

Every customer has an `iot:Data-ATS` and an `iot:Data` endpoint. Each endpoint uses an X.509 certificate to authenticate the client. We strongly recommend that customers use the newer `iot:Data-ATS` endpoint type to avoid issues related to the widespread distrust of Symantec certificate authorities. We provide the `iot:Data` endpoint for devices to retrieve data from old endpoints that use VeriSign certificates for backward compatibility. For more information, see Server Authentication.

AWS IoT endpoints for devices

Endpoint purpose	*`endpointType`* value	Description
AWS IoT Core - data plane operations	`iot:Data-ATS`	Used to send and receive data to and from the message broker, Device Shadow, and Rules Engine components of AWS IoT. `iot:Data-ATS` returns an ATS signed data endpoint.
AWS IoT Core - data plane operations (legacy)	`iot:Data`	`iot:Data` returns a VeriSign signed data endpoint provided for backward compatibility. MQTT 5 is not supported on Symantec (`iot:Data`) endpoints.
AWS IoT Core credential access	`iot:CredentialProvider`	Used to exchange a device's built-in X.509 certificate for temporary credentia

Endpoint purpose	*endpointType* value	Description
		ls to connect directly with other AWS services. For more information about connectin g to other AWS services, see Authorizing Direct Calls to AWS Services.
AWS IoT Device Managemen t - jobs data operations	`iot:Jobs`	Used to enable devices to interact with the AWS IoT Jobs service using the Jobs Device HTTPS APIs. `iot:Jobs` can be used for IPv4 only. If you are using dual-stack endpoints (IPv4 and IPv6), use the `iot:Data-ATS` endpoint type.
AWS IoT Device Advisor operations	`iot:DeviceAdvisor`	A test endpoint type used for testing devices with Device Advisor. For more informati on, see ???.
AWS IoT Core data beta (preview)	`iot:Data-Beta`	An endpoint type reserved for beta releases. For information about its current use, see ???.

You can also use your own fully-qualified domain name (FQDN), such as *`example.com`*, and the associated server certificate to connect devices to AWS IoT by using the section called "Domain configurations".

AWS IoT Device SDKs

The AWS IoT Device SDKs help you connect your IoT devices to AWS IoT Core and they support MQTT and MQTT over WSS protocols.

The AWS IoT Device SDKs differ from the AWS SDKs in that the AWS IoT Device SDKs support the specialized communications needs of IoT devices, but don't support all of the services supported by the AWS SDKs. The AWS IoT Device SDKs are compatible with the AWS SDKs that support all of the AWS services; however, they use different authentication methods and connect to different endpoints, which could make using the AWS SDKs impractical on an IoT device.

Mobile devices

The the section called "AWS Mobile SDKs" support both MQTT device communications, some of the AWS IoT service APIs, and the APIs of other AWS services. If you're developing on a supported mobile device, review its SDK to see if it's the best option for developing your IoT solution.

C++

AWS IoT C++ Device SDK

The AWS IoT C++ Device SDK allows developers to build connected applications using AWS and the APIs of the AWS IoT Core services. Specifically, this SDK was designed for devices that are not resource constrained and require advanced features such as message queuing, multi-threading support, and the latest language features. For more information, see the following:

- AWS IoT Device SDK C++ v2 on GitHub
- AWS IoT Device SDK C++ v2 Readme
- AWS IoT Device SDK C++ v2 Samples
- AWS IoT Device SDK C++ v2 API documentation

Python

AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python makes it possible for developers to write Python scripts to use their devices to access the AWS IoT platform through MQTT or MQTT over the WebSocket Secure (WSS) protocol. By connecting their devices to the APIs of the AWS IoT Core services, users can securely work with the message broker, rules, and Device Shadow service that AWS IoT Core provides and with other AWS services like AWS Lambda, Amazon Kinesis, and Amazon S3, and more.

- AWS IoT Device SDK for Python v2 on GitHub
- AWS IoT Device SDK for Python v2 Readme

- AWS IoT Device SDK for Python v2 Samples
- AWS IoT Device SDK for Python v2 API documentation

JavaScript

AWS IoT Device SDK for JavaScript

The AWS IoT Device SDK for JavaScript makes it possible for developers to write JavaScript applications that access APIs of the AWS IoT Core using MQTT or MQTT over the WebSocket protocol. It can be used in Node.js environments and browser applications. For more information, see the following:

- AWS IoT Device SDK for JavaScript v2 on GitHub
- AWS IoT Device SDK for JavaScript v2 Readme
- AWS IoT Device SDK for JavaScript v2 Samples
- AWS IoT Device SDK for JavaScript v2 API documentation

Java

AWS IoT Device SDK for Java

The AWS IoT Device SDK for Java makes it possible for Java developers to access the APIs of the AWS IoT Core through MQTT or MQTT over the WebSocket protocol. The SDK supports the Device Shadow service. You can access shadows by using HTTP methods, including GET, UPDATE, and DELETE. The SDK also supports a simplified shadow access model, which allows developers to exchange data with shadows by using getter and setter methods, without having to serialize or deserialize any JSON documents. For more information, see the following:

- AWS IoT Device SDK for Java v2 on GitHub
- AWS IoT Device SDK for Java v2 Readme
- AWS IoT Device SDK for Java v2 Samples
- AWS IoT Device SDK for Java v2 API documentation

Embedded C

AWS IoT Device SDK for Embedded C

> **Important**
>
> This SDK is intended for use by experienced embedded-software developers.

The AWS IoT Device SDK for Embedded C (C-SDK) is a collection of C source files under the MIT open source license that can be used in embedded applications to securely connect IoT devices to AWS IoT Core. It includes MQTT, JSON Parser, and AWS IoT Device Shadow libraries and others. It is distributed in source form and intended to be built into customer firmware along with application code, other libraries and, optionally, an RTOS (Real Time Operating System).

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs). If your device has sufficient memory and processing resources available, we recommend that you use one of the other AWS IoT Device and Mobile SDKs, such as the AWS IoT Device SDK for C++, Java, JavaScript, or Python.

For more information, see the following:

- AWS IoT Device SDK for Embedded C on GitHub
- AWS IoT Device SDK for Embedded C Readme
- AWS IoT Device SDK for Embedded C Samples

Device communication protocols

AWS IoT Core supports devices and clients that use the MQTT and the MQTT over WebSocket Secure (WSS) protocols to publish and subscribe to messages, and devices and clients that use the HTTPS protocol to publish messages. All protocols support IPv4 and IPv6. This section describes the different connection options for devices and clients.

TLS protocol versions

AWS IoT Core uses TLS version 1.2 and TLS version 1.3 to encrypt all communication. You can configure additional TLS policy versions for your endpoint by configuring TLS settings in domain configurations. When connecting devices to AWS IoT Core, clients can send the Server Name Indication (SNI) extension, which is required for features such as multi-account registration, configurable endpoints, custom domains, and VPC endpoints. For more information, see Transport Security in AWS IoT.

The AWS IoT Device SDKs support MQTT and MQTT over WSS and support the security requirements of client connections. We recommend using the AWS IoT Device SDKs to connect clients to AWS IoT.

Protocols, port mappings, and authentication

How a device or client connects to the message broker is configurable using an authentication type. By default or when no SNI extension is sent, authentication method is based on application protocol, port, and Application Layer Protocol Negotiation (ALPN) TLS extension that devices use. The following table lists the authentication expected based on port, port, and ALPN.

Protocols, authentication, and port mappings

Protocol	Operations supported	Authentication	Port	ALPN protocol name
MQTT over WebSocket	Publish, Subscribe	Signature Version 4	443	N/A
MQTT over WebSocket	Publish, Subscribe	Custom authentication	443	N/A
MQTT	Publish, Subscribe	X.509 client certificate	443†	`x-amzn-mqtt-ca`
MQTT	Publish, Subscribe	X.509 client certificate	8883	N/A
MQTT	Publish, Subscribe	Custom authentication	443†	`mqtt`
HTTPS	Publish only	Signature Version 4	443	N/A
HTTPS	Publish only	X.509 client certificate	443†	`x-amzn-http-ca`

Protocol	Operations supported	Authentication	Port	ALPN protocol name
HTTPS	Publish only	X.509 client certificate	8443	N/A
HTTPS	Publish only	Custom authentication	443	N/A

Application Layer Protocol Negotiation (ALPN)

†When using default endpoint configurations, clients that connect on port 443 with X.509 client certificate authentication must implement the Application Layer Protocol Negotiation (ALPN) TLS extension and use the ALPN protocol name listed in the ALPN ProtocolNameList sent by the client as part of the `ClientHello` message.
On port 443, the IoT:Data-ATS endpoint supports ALPN x-amzn-http-ca HTTP, but the IoT:Jobs endpoint does not.
On port 8443 HTTPS and port 443 MQTT with ALPN x-amzn-mqtt-ca, custom authentication can't be used.

Clients connect to their AWS account's device endpoints. See the section called “AWS IoT device data and service endpoints” for information about how to find your account's device endpoints.

Note

AWS SDKs don't require the entire URL. They only require the endpoint hostname such as the `pubsub.py` sample for AWS IoT Device SDK for Python on GitHub. Passing the entire URL as provided in the following table can generate an error such as invalid hostname.

Connecting to AWS IoT Core

Protocol	Endpoint or URL
MQTT	*`iot-endpoint`*

Protocol	Endpoint or URL
MQTT over WSS	`wss://`*`iot-endpoint`*` /mqtt`
HTTPS	`https://`*`iot-endpoint`*` /topics`

Choosing an application protocol for your device communication

For most IoT device communication through the device endpoints, you'll want to use the Secure MQTT or MQTT over WebSocket Secure (WSS) protocols; however, the device endpoints also support HTTPS.

The following table compares how AWS IoT Core uses the two high-level protocols (MQTT and HTTPS) for device communication.

AWS IoT device protocols (MQTT and HTTPS) side-by-side

Feature	MQTT	HTTPS
Publish/Subscribe support	Publish and subscribe	Publish only
SDK support	AWS Device SDKs support MQTT and WSS protocols	No SDK support, but you can use language-specific methods to make HTTPS requests
Quality of Service support	MQTT QoS levels 0 and 1	QoS is supported by passing a query string parameter `?qos=qos` where the value can be 0 or 1. You can add this query string to publish a message with the QoS value you want.
Can receive messages be missed while device was offline	Yes	No
`clientId` field support	Yes	No

Feature	MQTT	HTTPS
Device disconnection detection	Yes	No
Secure communications	Yes. See ???	Yes. See ???
Topic definitions	Application defined	Application defined
Message data format	Application defined	Application defined
Protocol overhead	Lower	Higher
Power consumption	Lower	Higher

Choosing an authentication type for your device communication

You can configure authentication type for your IoT endpoint using configurable endpoints. Alternatively, use default configuration and determine how your devices authenticate with application protocol, port, and ALPN TLS extension combination. The authentication type you choose determines how your devices will authenticate when connecting to AWS IoT Core. There are five authentication types:

X.509 certificate

Authenticate devices using X.509 client certificates, which AWS IoT Core validates to authenticate the device. This authentication type works with Secure MQTT (MQTT over TLS) and HTTPS protocols.

X.509 certificate with custom authorizer

Authenticate devices using X.509 client certificates and perform additional authentication actions using a custom authorizer, which will receive X.509 client certificate information. This authentication type works with Secure MQTT (MQTT over TLS) and HTTPS protocols. This authentication type is only possible using configurable endpoints with X.509 custom authentication. There is no ALPN option.

AWS Signature Version 4 (SigV4)

Authenticate devices using Cognito or your backend service, supporting social and enterprise federation. This authentication type works with MQTT over WebSocket Secure (WSS) and HTTPS protocols.

Custom authorizer

Authenticate devices by configuring a Lambda function to process custom authentication information sent to AWS IoT Core. This authentication type works with Secure MQTT (MQTT over TLS) , HTTPS, and MQTT over WebSocket Secure (WSS) protocols.

Default

Authenticate devices based on the port and/or application layer protocol negotiation (ALPN) extension that devices use. Some additional authentication options are not supported. For more information, see ???.

The table below shows all the supported combinations of authentication types and application protocols.

Supported combinations of authentication types and application protocols

Authentication type	Secure MQTT (MQTT over TLS)	MQTT over WebSocket Secure (WSS)	HTTPS	Default
X.509 certificate	✓		✓	
X.509 certificate with custom authorizer	✓		✓	
AWS Signature Version 4 (SigV4)		✓	✓	
Custom authorizer	✓	✓	✓	
Default				✓

Connection duration limits

HTTPS connections aren't guaranteed to last any longer than the time it takes to receive and respond to requests.

MQTT connection duration depends on the authentication feature that you use. The following table lists the maximum connection duration under ideal conditions for each feature.

MQTT connection duration by authentication feature

Feature	Maximum duration *
X.509 client certificate	1–2 weeks
Custom authentication	1–2 weeks
Signature Version 4	Up to 24 hours

* Not guaranteed

With X.509 certificates and custom authentication, connection duration has no hard limit, but it can be as short as a few minutes. Connection interruptions can occur for various reasons. The following list contains some of the most common reasons.

- Wi-Fi availability interruptions
- Internet service provider (ISP) connection interruptions
- Service patches
- Service deployments
- Service auto scaling
- Unavailable service host
- Load balancer issues and updates
- Client-side errors

Your devices must implement strategies for detecting disconnections and reconnecting. For information about disconnect events and guidance on how to handle them, see ??? in ???.

MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight and widely adopted messaging protocol that is designed for constrained devices. AWS IoT Core support for MQTT is based on the MQTT v3.1.1 specification and the MQTT v5.0 specification, with some differences, as documented in the section called "AWS IoT differences from MQTT specifications". As the latest version of the standard, MQTT 5 introduces several key features that make an MQTT-based system more robust, including new scalability enhancements, improved error reporting with reason code responses, message and session expiry timers, and custom user message headers. For more information about MQTT 5 features that AWS IoT Core supports, see MQTT 5 supported features. AWS IoT Core also supports cross MQTT version (MQTT 3 and MQTT 5) communication. An MQTT 3 publisher can send an MQTT 3 message to an MQTT 5 subscriber that will be receiving an MQTT 5 publish message, and vice versa.

AWS IoT Core supports device connections that use the MQTT protocol and MQTT over WSS protocol and that are identified by a *client ID*. The AWS IoT Device SDKs support both protocols and are the recommended ways to connect devices to AWS IoT Core. The AWS IoT Device SDKs support the functions necessary for devices and clients to connect to and access AWS IoT services. The Device SDKs support the authentication protocols that the AWS IoT services require and the connection ID requirements that the MQTT protocol and MQTT over WSS protocols require. For information about how to connect to AWS IoT using the AWS Device SDKs and links to examples of AWS IoT in the supported languages, see the section called "Connecting with MQTT using the AWS IoT Device SDKs". For more information about authentication methods and the port mappings for MQTT messages, see ???.

While we recommend using the AWS IoT Device SDKs to connect to AWS IoT, they are not required. If you do not use the AWS IoT Device SDKs, however, you must provide the necessary connection and communication security. Clients must send the Server Name Indication (SNI) TLS extension in the connection request. Connection attempts that don't include the SNI are refused. For more information, see Transport Security in AWS IoT. Clients that use IAM users and AWS credentials to authenticate clients must provide the correct Signature Version 4 authentication.

After your clients are connected, you can monitor and manage their MQTT client connections using APIs. For more information, see the section called "Managing MQTT connections".

In this topic:

- Connecting with MQTT using the AWS IoT Device SDKs
- MQTT Quality of Service (QoS) options

- MQTT persistent sessions
- MQTT retained messages
- MQTT Last Will and Testament (LWT) messages
- Using connectAttributes
- MQTT 5 supported features
- MQTT 5 properties
- MQTT reason codes
- AWS IoT differences from MQTT specifications
- Managing MQTT connections

Connecting with MQTT using the AWS IoT Device SDKs

This section contains links to the AWS IoT Device SDKs and to the source code of sample programs that illustrate how to connect a device to AWS IoT. The sample apps linked here show how to connect to AWS IoT using the MQTT protocol and MQTT over WSS.

> **Note**
>
> The AWS IoT Device SDKs have released an MQTT 5 client.

C++

Using the AWS IoT C++ Device SDK to connect devices

- Source code of a sample app that shows an MQTT connection example in C++
- AWS IoT Device SDK for C++ v2 on GitHub

Python

Using the AWS IoT Device SDK for Python to connect devices

- Source code of a sample app that shows an MQTT connection example in Python
- AWS IoT Device SDK v2 for Python on GitHub

JavaScript

Using the AWS IoT Device SDK for JavaScript to connect devices

- Source code of a sample app that shows an MQTT connection example in JavaScript
- AWS IoT Device SDK for JavaScript v2 on GitHub

Java

Using the AWS IoT Device SDK for Java to connect devices

> **Note**
>
> The AWS IoT Device SDK for Java v2 now supports Android development. For more information, see AWS IoT Device SDK for Android.

- Source code of a sample app that shows an MQTT connection example in Java
- AWS IoT Device SDK for Java v2 on GitHub

Embedded C

Using the AWS IoT Device SDK for Embedded C to connect devices

> **Important**
>
> This SDK is intended for use by experienced embedded-software developers.

- Source code of a sample app that shows an MQTT connection example in Embedded C
- AWS IoT Device SDK for Embedded C on GitHub

MQTT Quality of Service (QoS) options

AWS IoT and the AWS IoT Device SDKs support the MQTT Quality of Service (QoS) levels `0` and 1. The MQTT protocol defines a third level of QoS, level 2, but AWS IoT does not support it. Only the MQTT protocol supports the QoS feature. HTTPS supports QoS by passing a query string parameter `?qos=qos` where the value can be 0 or 1.

This table describes how each QoS level affects messages published to and by the message broker.

With a QoS level of...	The message is...	Comments
QoS level 0	Sent zero or more times	This level should be used for messages that are sent over reliable communication links or that can be missed without a problem.
QoS level 1	Sent at least one time, and then repeatedly until a PUBACK response is received	The message is not considere d complete until the sender receives a PUBACK response to indicate successful delivery.

MQTT persistent sessions

Persistent sessions store a client's subscriptions and messages, with a Quality of Service (QoS) of 1, that haven't been acknowledged by the client. When the device reconnects to a persistent session, the session resumes, subscriptions are reinstated, and unacknowledged subscribed messages received and stored prior to the reconnection are sent to the client.

The processing of the stored messages is recorded in CloudWatch metrics and CloudWatch Logs. For information about the entries written to CloudWatch and CloudWatch Logs, see Message broker metrics and Queued log entry.

Creating a persistent session

In MQTT 3, you create an MQTT persistent session by sending a `CONNECT` message and setting the `cleanSession` flag to `0`. If no session exists for the client sending the `CONNECT` message, a new persistent session is created. If a session already exists for the client, the client resumes the existing session. To create a clean session, you send a `CONNECT` message and set the `cleanSession` flag to 1, and the broker will not store any session state when the client disconnects.

In MQTT 5, you handle persistent sessions by setting the `Clean Start` flag and `Session Expiry Interval`. Clean Start controls the beginning of the connecting session and the end of the previous session. When you set `Clean Start` = `1`, a new session is created and a previous session is terminated if it exists. When you set `Clean Start`= `0`, the connecting session resumes a previous session if it exists. Session Expiry Interval controls the end of the connecting session. Session Expiry Interval specifies the time, in seconds (4-byte integer), that a session will persist after disconnect. Setting `Session Expiry interval=0` causes the session to terminate immediately upon disconnect. If the Session Expiry Interval is not specified in the CONNECT message, the default is 0.

MQTT 5 Clean Start and Session Expiry

Property value	Description
`Clean Start= 1`	Creates a new session and terminates a previous session if one exists.
`Clean Start= 0`	Resumes a session if a previous session exists.
`Session Expiry Interval> 0`	Persists a session.
`Session Expiry interval= 0`	Does not persist a session.

In MQTT 5, if you set `Clean Start` = `1` and `Session Expiry Interval` = `0`, this is the equivalent of an MQTT 3 clean session. If you set `Clean Start` = `0` and `Session Expiry Interval`> `0`, this is the equivalent of an MQTT 3 persistent session.

Note

Cross MQTT version (MQTT 3 and MQTT 5) persistent sessions are not supported. An MQTT 3 persistent session can't be resumed as an MQTT 5 session, and vice versa.

Operations during a persistent session

Clients use the `sessionPresent` attribute in the connection acknowledged (`CONNACK`) message to determine if a persistent session is present. If `sessionPresent` is `1`, a persistent session is present and any stored messages for the client are delivered to the client after the client receives the `CONNACK`, as described in Message traffic after reconnection to a persistent session. If `sessionPresent` is `1`, the client does not need to resubscribe. However, if `sessionPresent` is `0`, no persistent session is present and the client must resubscribe to its topic filters.

After the client joins a persistent session, it can publish messages and subscribe to topic filters without any additional flags on each operation.

Message traffic after reconnection to a persistent session

A persistent session represents an ongoing connection between a client and an MQTT message broker. When a client connects to the message broker using a persistent session, the message broker saves all subscriptions that the client makes during the connection. When the client disconnects, the message broker stores unacknowledged QoS 1 messages and new QoS 1 messages published to topics to which the client is subscribed. Messages are stored according to account limit. Messages that exceed the limit will be dropped. For more information about persistent message limits, see AWS IoT Core endpoints and quotas. When the client reconnects to its persistent session, all subscriptions are reinstated and all stored messages are sent to the client at a maximum rate of 10 messages per second. In MQTT 5, if an outbound QoS 1 with the Message Expiry Interval expires when a client is offline, after the connection resumes, the client won't receive the expired message.

After reconnection, the stored messages are sent to the client, at a rate that is limited to 10 stored messages per second, along with any current message traffic until the `Publish requests per second per connection` limit is reached. Because the delivery rate of the stored messages is limited, it will take several seconds to deliver all stored messages if a session has more than 10 stored messages to deliver after reconnection.

For shared subscribers, messages will be queued if at least one subscriber of a group uses a persistent session and no subscribers are online to receive the QoS 1 message. Dequeuing of messages is done at a maximum speed of 20 messages per second per active subscriber in a group. For more information, see shared subscriptions message queuing.

Ending a persistent session

Persistent sessions can end in the following ways:

- The persistent session expiration time elapses. The persistent session expiration timer starts when the message broker detects that a client has disconnected, either by the client disconnecting or the connection timing out.
- The client sends a `CONNECT` message that sets the `cleanSession` flag to `1`.
- You manually disconnect the client and clear the session using `DeleteConnection` API. For more information, see the section called "Managing MQTT connections".

In MQTT 3, the default value of persistent sessions expiration time is an hour, and this applies to all the sessions in the account.

In MQTT 5, you can set the Session Expiry Interval for each session on CONNECT and DISCONNECT packets.

For Session Expiry Interval on DISCONNECT packet:

- If the current session has a Session Expiry Interval of 0, you can't set Session Expiry Interval to greater than 0 on the DISCONNECT packet.
- If the current session has a Session Expiry Interval of greater than 0, and you set the Session Expiry Interval to 0 on the DISCONNECT packet, the session will be ended on DISCONNECT.
- Otherwise, the Session Expiry Interval on DISCONNECT packet will update the Session Expiry Interval of the current session.

> **Note**
>
> The stored messages waiting to be sent to the client when a session ends are discarded; however, they are still billed at the standard messaging rate, even though they could not be sent. For more information about message pricing, see AWS IoT Core Pricing. You can configure the expiration time interval.

Reconnection after a persistent session has expired

If a client doesn't reconnect to its persistent session before it expires, the session ends and its stored messages are discarded. When a client reconnects after the session has expired with a `cleanSession` flag to `0`, the service creates a new persistent session. Any subscriptions or messages from the previous session are not available to this session because they were discarded when the previous session expired.

Persistent session message charges

Messages are charged to your AWS account when the message broker sends a message to a client or an offline persistent session. When an offline device with a persistent session reconnects and resumes its session, the stored messages are delivered to the device and charged to your account again. For more information about message pricing, see AWS IoT Core pricing - Messaging.

The default persistent session expiration time of one hour can be increased by using the standard limit increase process. Note that increasing the session expiration time might increase your message charges because the additional time could allow for more messages to be stored for the offline device and those additional messages would be charged to your account at the standard messaging rate. The session expiration time is approximate and a session could persist for up to 30 minutes longer than the account limit; however, a session will not be shorter than the account limit. For more information about session limits, see AWS Service Quotas.

MQTT retained messages

AWS IoT Core supports the `RETAIN` flag described in the MQTT protocol. When a client sets the `RETAIN` flag on an MQTT message that it publishes, AWS IoT Core saves the message. It can then be sent to new subscribers, retrieved by calling the `GetRetainedMessage` operation, and viewed in the AWS IoT console.

Examples of using MQTT retained messages

- **As an initial configuration message**

  MQTT retained messages are sent to a client after the client subscribes to a topic. If you want all clients that subscribe to a topic to receive the MQTT retained message right after their subscription, you can publish a configuration message with the `RETAIN` flag set. Subscribing clients also receive updates to that configuration whenever a new configuration message is published.
- **As a last-known state message**

  Devices can set the `RETAIN` flag on current-state messages so that AWS IoT Core will save them. When applications connect or reconnect, they can subscribe to this topic and get the last reported state right after subscribing to the retained message topic. This way they can avoid having to wait until the next message from the device to see the current state.

In this section:

- Common tasks with MQTT retained messages in AWS IoT Core
- Billing and retained messages
- Comparing MQTT retained messages and MQTT persistent sessions
- MQTT retained messages and AWS IoT Device Shadows

Common tasks with MQTT retained messages in AWS IoT Core

AWS IoT Core saves MQTT messages with the `RETAIN` flag set. These *retained messages* are sent to all clients that have subscribed to the topic, as a normal MQTT message, and they are also stored to be sent to new subscribers to the topic.

MQTT retained messages require specific policy actions to authorize clients to access them. For examples of using retained message policies, see Retained message policy examples.

This section describes common operations that involve retained messages.

- **Creating a retained message**

  The client determines whether a message is retained when it publishes an MQTT message. Clients can set the `RETAIN` flag when they publish a message by using a Device SDK. Applications and services can set the `RETAIN` flag when they use the `Publish` action to publish an MQTT message.

  Only one message per topic name is retained. A new message with the RETAIN flag set published to a topic replaces any existing retained message that was sent to the topic earlier.

> **Note**
>
> You can't publish to a reserved topic with the `RETAIN` flag set.

- **Subscribing to a retained message topic**

  Clients subscribe to retained message topics as they would any other MQTT message topic. Retained messages received by subscribing to a retained message topic have the `RETAIN` flag set.

  Retained messages are deleted from AWS IoT Core when a client publishes a retained message with a 0-byte message payload to the retained message topic. Clients that have subscribed to the retained message topic will also receive the 0-byte message.

  Subscribing to a wild card topic filter that includes a retained message topic lets the client receive subsequent messages published to the retained message's topic, but it doesn't deliver the retained message upon subscription.

> **Note**
>
> To receive a retained message upon subscription, the topic filter in the subscription request must match the retained message topic exactly.

Retained messages received upon subscribing to a retained message topic have the `RETAIN` flag set. Retained messages that are received by a subscribing client after subscription, don't.

- **Retrieving a retained message**

  Retained messages are delivered to clients automatically when they subscribe to the topic with the retained message. For a client to receive the retained message upon subscription, it must subscribe to the exact topic name of the retained message. Subscribing to a wild card topic filter that includes a retained message topic lets the client receive subsequent messages published to the retained message's topic, but it does not deliver the retained message upon subscription.

  Services and apps can list and retrieve retained messages by calling [`ListRetainedMessages`]() and [`GetRetainedMessage`]().

  A client is not prevented from publishing messages to a retained message topic *without* setting the `RETAIN` flag. This could cause unexpected results, such as the retained message not matching the message received by subscribing to the topic.

  With MQTT 5, if a retained message has the Message Expiry Interval set and the retained message expires, a new subscriber that subscribes to that topic will not receive the retained message upon successful subscription.

- **Listing retained message topics**

  You can list retained messages by calling [`ListRetainedMessages`]() and the retained messages can be viewed in the [AWS IoT console]().

- **Getting retained message details**

  You can get retained message details by calling [`GetRetainedMessage`]() and they can be viewed in the [AWS IoT console]().

- **Retaining a Will message**

MQTT *Will* messages that are created when a device connects can be retained by setting the `Will Retain` flag in the `Connect Flag bits` field.

- **Deleting a retained message**

  Devices, applications, and services can delete a retained message by publishing a message with the `RETAIN` flag set and an empty (0-byte) message payload to the topic name of the retained message to delete. Such messages delete the retained message from AWS IoT Core, are sent to clients with a subscription to the topic, but they are not retained by AWS IoT Core.

  Retained messages can also be deleted interactively by accessing the retained message in the AWS IoT console. Retained messages that are deleted by using the AWS IoT console also send a 0-byte message to clients that have subscribed to the retained message's topic.

  Retained messages can't be restored after they are deleted. A client would need to publish a new retained message to take the place of the deleted message.

- **Debugging and troubleshooting retained messages**

  The AWS IoT console provides several tools to help you troubleshoot retained messages:

  - **The Retained messages page**

    The **Retained messages** page in the AWS IoT console provides a paginated list of the retained messages that have been stored by your Account in the current Region. From this page, you can:

    - See the details of each retained message, such as the message payload, QoS, the time it was received.
    - Update the contents of a retained message.
    - Delete a retained message.

  - **The MQTT test client**

    The **MQTT test client** page in the AWS IoT console can subscribe and publish to MQTT topics. The publish option lets you set the RETAIN flag on the messages that you publish to simulate how your devices might behave. You can also use the MQTT test client to monitor messages from connected clients that you manage through the client connections interface. For more information about managing client connections, see the section called “Managing MQTT connections”.

Some unexpected results might be the result of these aspects of how retained messages are implemented in AWS IoT Core.

- **Retained message limits**

  When an account has stored the maximum number of retained messages, AWS IoT Core returns a throttled response to messages published with RETAIN set and payloads greater than 0 bytes until some retained messages are deleted and the retained message count falls below the limit.

- **Retained message delivery order**

  The sequence of retained message and subscribed message delivery is not guaranteed.

Billing and retained messages

Publishing messages with the `RETAIN` flag set from a client, by using AWS IoT console, or by calling `Publish` incurs additional messaging charges described in AWS IoT Core pricing - Messaging.

Retrieving retained messages by a client, by using AWS IoT console, or by calling `GetRetainedMessage` incurs messaging charges in addition to the normal API usage charges. The additional charges are described in AWS IoT Core pricing - Messaging.

MQTT *Will* messages that are published when a device disconnects unexpectedly incur messaging charges described in AWS IoT Core pricing - Messaging.

For more information about messaging costs, see AWS IoT Core pricing - Messaging.

Comparing MQTT retained messages and MQTT persistent sessions

Retained messages and persistent sessions are standard features of MQTT that make it possible for devices to receive messages that were published while they were offline. Retained messages can be published from persistent sessions. This section describes key aspects of these features and how they work together.

	Retained messages	Persistent sessions
Key features	Retained messages can be used to configure or notify	Persistent sessions are useful for devices that have

	Retained messages	Persistent sessions
	large groups of devices after they connect. Retained messages can also be used where you want devices to receive only the last message published to a topic after a reconnection.	intermittent connectivity and could miss several important messages. Devices can connect with a persistent session to receive messages sent while they are offline.
Examples	Retained messages can give devices configuration information about their environment when they come online. The initial configura tion could include a list of other message topics to which it should subscribe or information about how it should configure its local time zone.	Devices that connect over a cellular network with intermittent connectivity could use persistent sessions to avoid missing important messages that are sent while a device is out of network coverage or needs to turn off its cellular radio.
Messages received on initial subscription to a topic	After subscribing to a topic with a retained message, the most recent retained message is received.	After subscribing to a topic without a retained message, no message is received until one is published to the topic.
Subscribed topics after reconnection	Without a persistent session, the client must subscribe to topics after reconnection.	Subscribed topics are restored after reconnection.
Messages received after reconnection	After subscribing to a topic with a retained message, the most recent retained message is received.	All messages published with a QOS = 1 and subscribe d to with a QOS =1 while the device was disconnec ted are sent after the device reconnects.

	Retained messages	Persistent sessions
Data/session expiration	In MQTT 3, retained messages do not expire. They are stored until they are replaced or deleted. In MQTT 5, retained messages expire after the Message Expiry Interval you set. For more information, see Message Expiry.	Persistent sessions expire if the client doesn't reconnect within the timeout period. After a persistent session expires, the client's subscriptions and saved messages that were published with a QOS = 1 and subscribed to with a QOS =1 while the device was disconnected are deleted. Expired messages won't be delivered. For more information about session expirations with persistent sessions, see the section called "MQTT persistent sessions".

For information about persistent sessions, see the section called "MQTT persistent sessions".

With Retained Messages, the publishing client determines whether a message should be retained and delivered to a device after it connects, whether it had a previous session or not. The choice to store a message is made by the publisher and the stored message is delivered to all current and future clients that subscribe to QoS 0 or QoS 1 subscriptions. Retained messages keep only one message on a given topic at a time.

When an account has stored the maximum number of retained messages, AWS IoT Core returns a throttled response to messages published with RETAIN set and payloads greater than 0 bytes until some retained messages are deleted and the retained message count falls below the limit.

MQTT retained messages and AWS IoT Device Shadows

Retained messages and Device Shadows both retain data from a device, but they behave differently and serve different purposes. This section describes their similarities and differences.

	Retained messages	Device Shadows
Message payload has a pre-defined structure or schema	As defined by the implement ation. MQTT does not specify a structure or schema for its message payload.	AWS IoT supports a specific data structure.
Updating the message payload generates event messages	Publishing a retained message sends the message to subscribed clients, but doesn't generate additional update messages.	Updating a Device Shadow produces update messages that describe the change.
Message updates are numbered	Retained messages are not numbered automatically.	Device Shadow documents have automatic version numbers and timestamps.
Message payload is attached to a thing resource	Retained messages are not attached to a thing resource.	Device Shadows are attached to a thing resource.
Updating individual elements of the message payload	Individual elements of the message can't be changed without updating the entire message payload.	Individual elements of a Device Shadow document can be updated without the need to update the entire Device Shadow document.
Client receives message data upon subscription	The client automatically receives a retained message after it subscribes to a topic with a retained message.	Clients can subscribe to Device Shadow updates, but they must request the current state deliberately.
Indexing and searchability	Retained messages are not indexed for search.	Fleet indexing indexes Device Shadow data for search and aggregation.

MQTT Last Will and Testament (LWT) messages

Last Will and Testament (LWT) is a feature in MQTT. With LWT, clients can specify a message which the broker will publish to a client-defined topic and send to all clients that subscribed to the topic when an uninitiated disconnection occurs. The message that clients specify is called an LWT message or a Will Message, and the topic that clients define is referred to as a Will Topic. You can specify an LWT message when a device connects to the broker. These messages can be retained by setting the `Will Retain` flag in the `Connect Flag bits` field during the connection. For example, if the `Will Retain` flag is set to `1`, a Will Message will be stored in the broker in the associated Will Topic. For more information, see Will Messages.

When managing client connections, you can control whether LWT messages are published when you disconnect a client. For more information, see the section called “Managing MQTT connections”.

The broker will store the Will Messages until an uninitiated disconnection occurs. When that happens, the broker will publish the messages to all clients that subscribed to the Will Topic to notify the disconnection. If the client disconnects from the broker with a client-initiated disconnection using the MQTT DISCONNECT message, the broker won't publish the stored LWT messages. In all other cases, the LWT messages will be dispatched. Due to the asynchronous nature of disconnect processing, LWT messages are not guaranteed to be dispatched in order during reconnection. We recommend that you use lifecycle events to improve the accuracy of connectivity state detection, as these events provide attributes such as timestamps and version numbers to manage out-of-order events. For a complete list of the disconnect scenarios when the broker will send the LWT messages, see Connect/Disconnect events.

Using connectAttributes

`ConnectAttributes` allow you to specify what attributes you want to use in your connect message in your IAM policies such as `PersistentConnect` and `LastWill`. With `ConnectAttributes`, you can build policies that don't give devices access to new features by default, which can be helpful if a device is compromised.

`connectAttributes` supports the following features:

`PersistentConnect`

Use the `PersistentConnect` feature to save all subscriptions the client makes during the connection when the connection between the client and broker is interrupted.

`LastWill`

Use the `LastWill` feature to publish a message to the `LastWillTopic` when a client unexpectedly disconnects.

By default, your policy has a non-persistent connection and there are no attributes passed for this connection. You must specify a persistent connection in your IAM policy if you want to have one.

When managing client connections, you can view the connection attributes and session configuration for connected clients. For more information, see the section called “Managing MQTT connections”.

For `ConnectAttributes` examples, see Connect Policy Examples.

MQTT 5 supported features

AWS IoT Core support for MQTT 5 is based on the MQTT v5.0 specification with some differences as documented in the section called “AWS IoT differences from MQTT specifications”.

AWS IoT Core supports the following MQTT 5 features:

- Shared subscriptions
- Clean Start and Session Expiry
- Reason code on all ACKs
- Topic aliases
- Message expiry
- Other MQTT 5 features

Shared subscriptions

AWS IoT Core supports shared subscriptions for both MQTT 3.1.1 and MQTT 5. Shared subscriptions allow multiple clients to share a subscription to a topic and only one client will receive messages published to that topic using a random distribution. Shared subscriptions can effectively load balance MQTT messages across a number of subscribers. For example, say you have 1,000 devices publishing to the same topic, and 10 backend applications processing those messages. In that case, the backend applications can subscribe to the same topic, and each would randomly receive messages published by the devices to the shared topic. This is effectively "sharing" the load of those messages. Shared subscriptions also allow for better resiliency. When any backend application disconnects, the broker distributes the load to remaining subscribers in the group. When all the subscribers disconnect, messages are queued.

Message queuing capabilities are available for shared subscriptions on both MQTT 3.1.1 and MQTT 5 connections to enhance message delivery reliability.

To use shared subscriptions, clients subscribe to a shared subscription's topic filter as follows:

```
$share/{ShareName}/{TopicFilter}
```

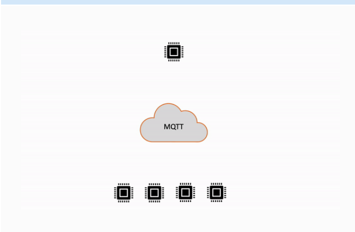
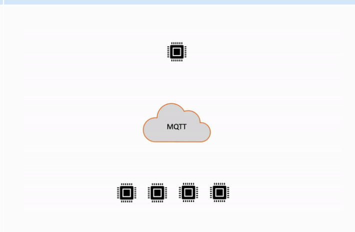
- `$share` is a literal string to indicate a Shared Subscription's topic filter, which must start with `$share`.
- `{ShareName}` is a character string to specify the shared name used by a group of subscribers. A shared subscription's topic filter must contain a `ShareName` and be followed by the / character. The `{ShareName}` must not include the following characters: /, +, or #. The maximum size for `{ShareName}` is 128 UTF-8 characters.
- `{TopicFilter}` follows the same topic filter syntax as a non-shared subscription. The maximum size for `{TopicFilter}` is 256 UTF-8 characters.
- The two required slashes (/) for `$share/{ShareName}/{TopicFilter}` are not included in the Maximum number of slashes in topic and topic filter limit.

Subscriptions that have the same `{ShareName}/{TopicFilter}` belong to the same shared subscription group. You can create multiple shared subscription groups and don't exceed the Shared Subscriptions per group limit. For more information, see AWS IoT Core endpoints and quotas from the *AWS General Reference*.

The following tables compare non-shared subscriptions and shared subscriptions:

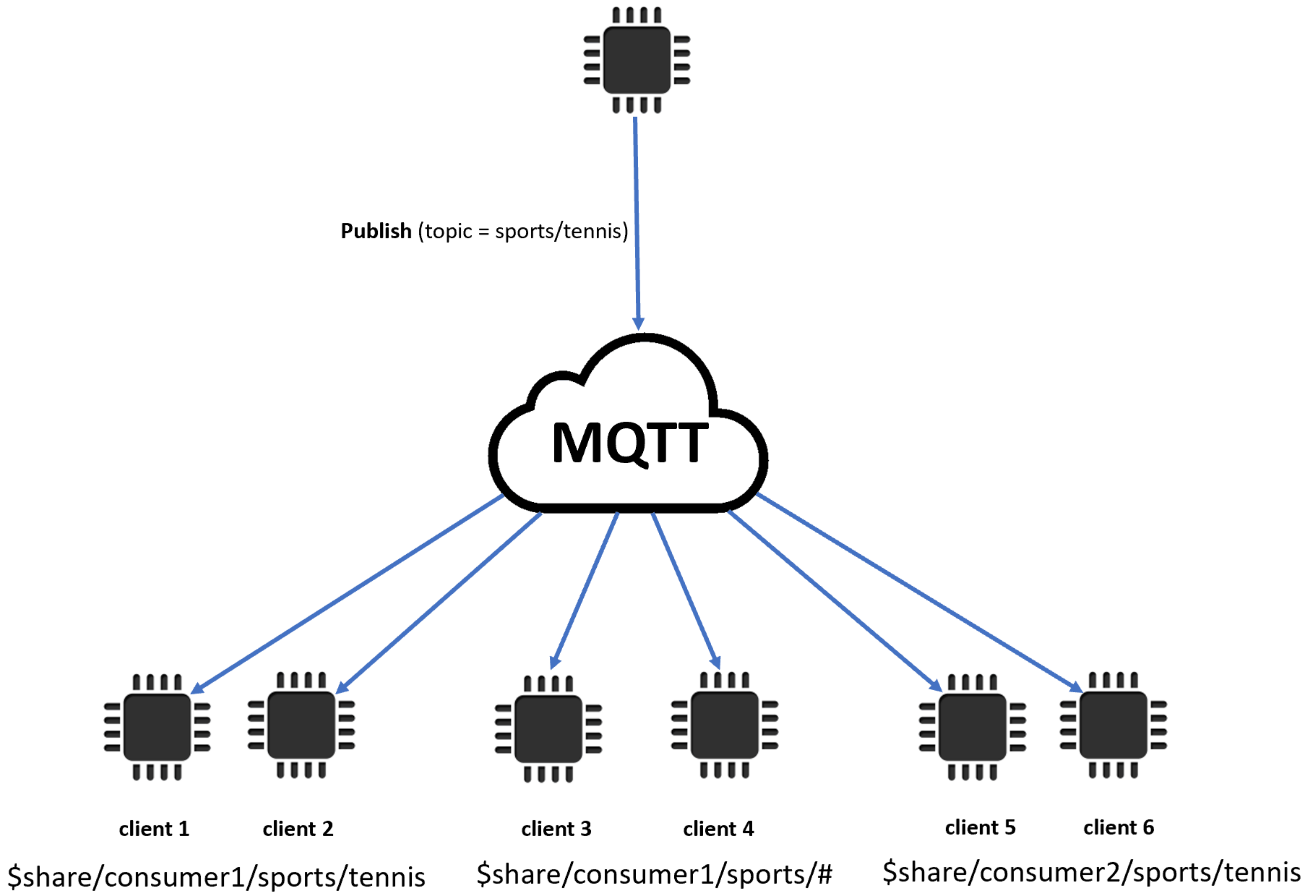
Subscription	Description	Topic filter examples
Non-shared subscriptions	Each client creates a separate subscription to receive the published messages. When a message is published to a topic, all subscribers to that topic receive a copy of the message.	`sports/tennis` `sports/#`

Subscription	Description	Topic filter examples
Shared subscriptions	Multiple clients can share a subscription to a topic and only one client will receive messages published to that topic at a random distribution.	`$share/consumer/sports/tennis` `$share/consumer/sports/#`

Non-shared subscriptions flow	Shared subscriptions flow		
			

Important notes for using shared subscriptions

- If the shared subscriber group consists of any persistent session subscribers, when all the subscribers in the shared group are disconnected, or if any subscribers breach the Publish requests per second per connection limit, any unacknowledged QoS 1 messages and undelivered QoS 1 messages published to a shared subscription group will be queued. For more information, see shared subscriptions message queuing.
- QoS 0 messages published to a shared subscription group will be dropped upon any failure.
- Shared subscriptions don't receive retained messages when subscribing to topic patterns as part of a shared subscriber group. Messages that are published on topics that have shared subscribers and have the `RETAIN` flag set are delivered to shared subscribers like any other publish message.
- When shared subscriptions contain wildcard characters (# or +), there might be multiple matching shared subscriptions to a topic. If that happens, the message broker copies the publishing message and sends it to a random client in each matching shared subscription. The wildcard behavior of shared subscriptions can be explained in the following diagram.



In this example, there are three matching shared subscriptions to the publishing MQTT topic `sports/tennis`. The message broker copies the published message and sends the message to a random client in each matching group.

Client 1 and client 2 share the subscription: `$share/consumer1/sports/tennis`

Client 3 and client 4 share the subscription: `$share/consumer1/sports/#`

Client 5 and client 6 share the subscription: `$share/consumer2/sports/tennis`

For more information about shared subscriptions limits, see [AWS IoT Core endpoints and quotas] from the *AWS General Reference*. To test shared subscriptions using the AWS IoT MQTT client in the [AWS IoT console], see [???]. You can also view which topics connected clients are subscribed to, including shared subscriptions, by using the client connection management features. For more information, see [the section called "Managing MQTT connections"]. For more information about shared subscriptions, see [Shared Subscriptions] from the MQTTv5.0 specification.

Shared subscriptions message queuing

To enhance message delivery reliability, shared subscriptions include message queuing capabilities that store messages when no online subscribers are available. When a shared subscription group contains at least one member with a persistent session, the queuing feature is enabled for the group. When distributing messages, online members are selected as recipients. QoS 1 messages are queued when no members are found online or when subscribers exceed the `Publish requests per second per connection` limit. Queued messages are delivered when either existing members resume their persistent sessions, or new members join the group. Queued messages are delivered at up to 20 queued messages per second per active group subscriber, along with any other messages delivered to the subscriber as per the subscriptions.

By default, queued message retention follows the `Persistent Session expiry period` quota. However, if a Message Expiry Interval (MEI) is set in the inbound publish message, the MEI takes precedence. When MEI is present, it determines the message retention period, regardless of the Persistent Session expiry period.

Message queue rates are limited according to the `Queued messages per second per account` quota, and the number of messages is limited by the `Maximum number of queued messages per shared subscription group` quota. To view and manage your quotas, access the Service Quotas console.

You can monitor the Queue CloudWatch by searching for `ApproximateQueueDepth` under the `AWS/Usage` namespace, or you can use the following CLI command to list the metrics associated with each shared subscription group's queue depth.

```
aws cloudwatch list-metrics --namespace "AWS/Usage" --dimensions
 Name=Resource,Value='ApproximateQueueDepth'
```

When these limits are exceeded, only the messages that were already queued before reaching the limit are retained. New incoming messages that would exceed the limits are dropped. The system does not replace older queued messages with newer ones.

The queuing of messages is recorded in CloudWatch metrics and CloudWatch Logs. For information about the entries written to CloudWatch and CloudWatch Logs, see Message broker metrics and Queued log entry. Queued messages are still billed at the standard messaging rate. For more information about message pricing, see AWS IoT Core Pricing.

Session lifecycle in shared subscription group

When a clean session subscribes to a group, it becomes an online member of the group. When it unsubscribes or disconnects, the clean session leaves the group.

When a persistent session subscribes to a group, it becomes an online member of the group. When it disconnects, it still remains in the group, but becomes an offline member of the group. When it reconnects, it becomes an online member again. The persistent session leaves the group when it explicitly unsubscribes or when it expires after disconnect.

Clean Start and Session Expiry

You can use Clean Start and Session Expiry to handle your persistent sessions with more flexibility. A Clean Start flag indicates whether the session should start without using an existing session. A Session Expiry interval indicates how long to retain the session after a disconnect. The session expiry interval can be modified at disconnect. For more information, see the section called “MQTT persistent sessions”.

Reason code on all ACKs

You can debug or process error messages more easily using the reason codes. Reason codes are returned by the message broker based on the type of interaction with the broker (Subscribe, Publish, Acknowledge). For more information, see MQTT reason codes. For a complete list of MQTT reason codes, see MQTT v5 specification.

Topic aliases

You can substitute a topic name with a topic alias, which is a two-byte integer. Using topic aliases can optimize the transmission of topic names to potentially reduce data costs on metered data services. AWS IoT Core has a default limit of 8 topic aliases. For more information, see AWS IoT Core endpoints and quotas from the *AWS General Reference*.

Message expiry

You can add message expiry values to published messages. These values represent the Message Expiry Interval (MEI) in seconds. If the messages haven't been sent to the subscribers within that interval, the message will expire and be removed. If you don't set the message expiry value, the message will not expire.

On the outbound, the subscriber will receive a message with the remaining time left in the expiry interval. For example, if an inbound publish message has a message expire of 30 seconds, and it's routed to the subscriber after 20 seconds, the message expiry field will be updated to 10. It is possible for the message received by the subscriber to have an updated MEI of 0. This is because as soon as the time remaining is 999 ms or less, it will be updated to 0.

In AWS IoT Core, the minimum MEI is 1. If the interval is set to 0 from the client side, it will be adjusted to 1. The maximum message expiry interval is 604800 (7 days). Any values higher than this will be adjusted to the maximum value.

In cross version communication, the behavior of message expiry is decided by MQTT version of the inbound publish message. For example, a message with message expiry sent by a session connected via MQTT5 can expire for devices subscribed with MQTT3 sessions. The table below lists how message expiry supports the following types of publish messages:

Publish Message Type	Message Expiry Interval
Regular Publish	If a server fails to deliver the message within the specified time, the expired message will be removed and the subscriber won't receive it. This includes situations such as when a device is not pubacking their QoS 1 messages.
Retain	If a retained message expires and a new client subscribes to the topic, the client won't receive the message upon subscription.
Last Will	The interval for last will messages starts after the client disconnects and the server attempts to deliver the last will message to its subscribers.
Queued messages	If an outbound QoS 1 with Message Expiry Interval expires when a client is offline, after the persistent session resumes, the client won't receive the expired message.

Other MQTT 5 features

Server disconnect

When a disconnection happens, the server can proactively send the client a DISCONNECT to notify connection closure with a reason code for disconnection.

Request/Response

Publishers can request a response be sent by the receiver to a publisher-specified topic upon reception.

Maximum Packet Size

Client and Server can independently specify the maximum packet size that they support.

Payload format and content type

You can specify the payload format (binary, text) and content type when a message is published. These are forwarded to the receiver of the message.

MQTT 5 properties

MQTT 5 properties are important additions to the MQTT standard to support new MQTT 5 features such as Session Expiry and the Request/Response pattern. In AWS IoT Core, you can create rules that can forward the properties in outbound messages, or use HTTP Publish to publish MQTT messages with some of the new properties.

The following table lists all the MQTT 5 properties that AWS IoT Core supports.

Property	Description	Input type	Packet
Payload Format Indicator	A Boolean value that indicates whether the payload is formatted as UTF-8.	Byte	PUBLISH, CONNECT
Content Type	A UTF-8 string that describes the content of the payload.	UTF-8 string	PUBLISH, CONNECT
Response Topic	A UTF-8 string that describes the topic the receiver should publish to as part of the request-response flow. The topic must not have wildcard characters.	UTF-8 string	PUBLISH, CONNECT
Correlation Data	Binary data used by the sender of the request message to identify which request the response message is for.	Binary	PUBLISH, CONNECT
User Property	A UTF-8 string pair. This property can appear multiple times in one packet. Receivers will	UTF-8 string pair	CONNECT, PUBLISH, Will Propertie s, SUBSCRIBE

Property	Description	Input type	Packet
	receive the key-value pairs in the same order they are sent.		, DISCONNECT, UNSUBSCRIBE
Message Expiry Interval	A 4-byte integer that represents the Message Expiry Interval in seconds. If absent, the message doesn't expire.	4-byte intege	PUBLISH, CONNECT
Session Expiry Interval	A 4-byte integer that represents the session expiry interval in seconds. AWS IoT Core supports a maximum of 7 days, with a default maximum of one hour. If the value you set exceeds the maximum of your account, AWS IoT Core will return the adjusted value in the CONNACK.	4-byte intege	CONNECT, CONNACK, DISCONNECT
Assigned Client Identifier	A random client ID generated by AWS IoT Core when a client ID isn't specified by devices. The random client ID must be a new client identifier that's not used by any other session currently managed by the broker.	UTF-8 string	CONNACK
Server Keep Alive	A 2-byte integer that represents the keep alive time assigned by the server. The server will disconnect the client if the client is inactive for more than the keep alive time.	2-byte intege	CONNACK
Request Problem Information	A Boolean value that indicates whether the Reason String or User Properties are sent in the case of failures.	Byte	CONNECT
Receive Maximum	A 2-byte integer that represents the maximum number of PUBLISH QOS > 0 packets which can be sent without receiving an PUBACK.	2-byte intege	CONNECT, CONNACK

Property	Description	Input type	Packet
Topic Alias Maximum	This value indicates the highest value that will be accepted as a Topic Alias. Default is 0.	2-byte intege	CONNECT, CONNACK
Maximum QoS	The maximum value of QoS that AWS IoT Core supports. Default is 1. AWS IoT Core doesn't support QoS 2.	Byte	CONNACK
Retain Available	A Boolean value that indicates whether AWS IoT Core message broker supports retained messages. The default is 1.	Byte	CONNACK
Maximum Packet Size	The maximum packet size that AWS IoT Core accepts and sends. Cannot exceed 128 KB.	4-byte intege	CONNECT, CONNACK
Wildcard Subscription Available	A Boolean value that indicates whether AWS IoT Core message broker supports Wildcard Subscription Available. The default is 1.	Byte	CONNACK
Subscript ion Identifier Available	A Boolean value that indicates whether AWS IoT Core message broker supports Subscript ion Identifier Available. The default is 0.	Byte	CONNACK

MQTT reason codes

MQTT 5 introduces improved error reporting with reason code responses. AWS IoT Core can return reason codes including but not limited to the following grouped by packets. For a complete list of reason codes supported by MQTT 5, see MQTT 5 specifications.

CONNACK Reason Codes

Value	Hex	Reason Code name	Description
0	0x00	Success	The connection is accepted.

Value	Hex	Reason Code name	Description
128	0x80	Unspecified error	The server does not wish to reveal the reason for the failure, or none of the other reason codes apply.
133	0x85	Client Identifier not valid	The client identifier is a valid string but is not allowed by the server.
134	0x86	Bad User Name or Password	The server does not accept the user name or password specified by the client.
135	0x87	Not authorized	The client is not authorized to connect.
144	0x90	Topic Name invalid	The Will Topic Name is correctly formed but is not accepted by the server.
151	0x97	Quota exceeded	An implementation or administrative imposed limit has been exceeded.
155	0x9B	QoS not supported	The server does not support the QoS set in Will QoS.

PUBACK Reason Codes

Value	Hex	Reason Code name	Description
0	0x00	Success	The message is accepted. Publication of the QoS 1 message proceeds.
128	0x80	Unspecified error	The receiver does not accept the publish, but either does not want to reveal the reason, or it does not match one of the other values.
135	0x87	Not authorize d	The PUBLISH is not authorized.

Value	Hex	Reason Code name	Description
144	0x90	Topic Name invalid	The topic name is not malformed, but is not accepted by the client or server.
145	0x91	Packet identifier in use	The packet identifier is already in use. This might indicate a mismatch in the session state between the client and server.
151	0x97	Quota exceeded	An implementation or administrative imposed limit has been exceeded.

DISCONNECT Reason Codes

Value	Hex	Reason Code name	Description
129	0x81	Malformed Packet	The received packet does not conform to this specifica tion.
130	0x82	Protocol Error	An unexpected or out of order packet was received.
135	0x87	Not authorize d	The request is not authorized.
139	0x8B	Server shutting down	The server is shutting down.
141	0x8D	Keep Alive timeout	The connection is closed because no packet has been received for 1.5 times the Keep Alive time.
142	0x8E	Session taken over	Another connection using the same ClientID has connected, causing this connection to be closed.
143	0x8F	Topic Filter invalid	The topic filter is correctly formed but is not accepted by the server.

Value	Hex	Reason Code name	Description
144	0x90	Topic Name invalid	The topic name is correctly formed but is not accepted by this client or server.
147	0x93	Receive Maximum exceeded	The client or server has received more than the Receive Maximum publication for which it has not sent PUBACK or PUBCOMP.
148	0x94	Topic Alias invalid	The client or server has received a PUBLISH packet containing a topic alias greater than the Maximum Topic Alias it sent in the CONNECT or CONNACK packet.
151	0x97	Quota exceeded	An implementation or administrative imposed limit has been exceeded.
152	0x98	Administrative action	The connection is closed due to an administrative action.
155	0x9B	QoS not supported	The client specified a QoS greater than the QoS specified in a Maximum QoS in the CONNACK.
161	0xA1	Subscription Identifiers not supported	The server does not support subscription identifiers; the subscription is not accepted.

SUBACK Reason Codes

Value	Hex	Reason Code name	Description
0	0x00	Granted QoS 0	The subscription is accepted and the maximum QoS sent will be QoS 0. This might be a lower QoS than was requested.

Value	Hex	Reason Code name	Description
1	0x01	Granted QoS 1	The subscription is accepted and the maximum QoS sent will be QoS 1. This might be a lower QoS than was requested.
128	0x80	Unspecified error	The subscription is not accepted and the Server either does not wish to reveal the reason or none of the other Reason Codes apply.
135	0x87	Not authorized	The Client is not authorized to make this subscription.
143	0x8F	Topic Filter invalid	The Topic Filter is correctly formed but is not allowed for this Client.
145	0x91	Packet Identifie r in use	The specified Packet Identifier is already in use.
151	0x97	Quota exceeded	An implementation or administrative imposed limit has been exceeded.

UNSUBACK Reason Codes

Value	Hex	Reason Code name	Description
0	0x00	Success	The subscription is deleted.
128	0x80	Unspecified error	The unsubscribe could not be completed and the Server either does not wish to reveal the reason or none of the other Reason Codes apply.
143	0x8F	Topic Filter invalid	The Topic Filter is correctly formed but is not allowed for this Client.
145	0x91	Packet Identifie r in use	The specified Packet Identifier is already in use.

AWS IoT differences from MQTT specifications

The message broker implementation is based on the MQTT v3.1.1 specification and the MQTT v5.0 specification, but it differs from the specifications in these ways:

- AWS IoT doesn't support the following packets for MQTT 3: PUBREC, PUBREL, and PUBCOMP.
- AWS IoT doesn't support the following packets for MQTT 5: PUBREC, PUBREL, PUBCOMP, and AUTH.
- AWS IoT doesn't support MQTT 5 server redirection.
- AWS IoT supports MQTT quality of service (QoS) levels 0 and 1 only. AWS IoT doesn't support publishing or subscribing with QoS level 2. When QoS level 2 is requested, the message broker doesn't send a PUBACK or SUBACK.
- In AWS IoT, subscribing to a topic with QoS level 0 means that a message is delivered zero or more times. A message might be delivered more than once. Messages delivered more than once might be sent with a different packet ID. In these cases, the DUP flag is not set.
- When responding to a connection request, the message broker sends a CONNACK message. This message contains a flag to indicate if the connection is resuming a previous session.
- Before sending additional control packets or a disconnect request, the client must wait for the CONNACK message to be received on their device from the AWS IoT message broker.
- When a client subscribes to a topic, there might be a delay between the time the message broker sends a SUBACK and the time the client starts receiving new matching messages.
- When a client uses the wildcard character # in the topic filter to subscribe to a topic, all strings at and below its level in the topic hierarchy are matched. However, the parent topic is not matched. For example, a subscription to the topic `sensor/#` receives messages published to the topics `sensor/`, `sensor/temperature`, `sensor/temperature/room1`, but not messages published to `sensor`. For more information about wildcards, see Topic name filters.
- The message broker uses the client ID to identify each client. The client ID is passed in from the client to the message broker as part of the MQTT payload. Two clients with the same client ID can't be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, the new client connection is accepted and the previously connected client is disconnected. You can also manually disconnect clients using APIs. For more information, see the section called “Managing MQTT connections”.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.

- Subscription to topic filters that contain a wildcard character can't receive retained messages. To receive a retained message, the subscribe request must contain a topic filter that matches the retained message topic exactly.
- The message broker doesn't guarantee the order in which messages and ACK are received.
- AWS IoT can have limits that are different from the specifications. For more information, see AWS IoT Core message broker and protocol limits and quotas from the *AWS IoT Reference Guide*.
- The MQTT DUP flag is not supported.

Managing MQTT connections

AWS IoT Core provides APIs to help you manage MQTT connections, including the ability to disconnect clients and manage their sessions. These capabilities give you more control over your AWS IoT client fleet and help with troubleshooting connection issues.

DeleteConnection API

Use the `DeleteConnection` API to disconnect MQTT devices from AWS IoT Core by specifying their client IDs. When you disconnect a client, AWS IoT Core disconnects the client from the AWS IoT Core message broker and can optionally clean the session state and suppress the Last Will and Testament (LWT) message.

When you call the `DeleteConnection` API, AWS IoT Core performs several actions to ensure a clean disconnection. AWS IoT Core first sends an MQTT disconnect message to the client to terminate the MQTT session. The service then closes the underlying TCP/TLS socket.

The message broker sends a `DISCONNECT` packet to the device and publishes a lifecycle event with the disconnect reason `API_INITIATED_DISCONNECT`. This helps you identify when a disconnection was initiated through the API rather than by the client or due to network issues. You can monitor these events for visibility, troubleshooting, and auditing purposes. For example, you can use AWS IoT rules to process these events to track when and why clients were disconnected.

If you set the `cleanSession` parameter to `true`, AWS IoT Core removes the client's session state, including all subscriptions and queued messages. If you clean a session, the persistent session is terminated. If the client was a persistent session and the `preventWillMessage` parameter is set to `false`, the service dispatches the LWT message if available, which is useful during planned maintenance operations.

When you call the `DeleteConnection` API, the disconnection process begins immediately, but the exact timing of when the client recognizes the disconnection can vary based on network conditions and client implementation. Most clients will detect the disconnection within a few seconds, but in some cases with poor network connectivity, it might take longer for the client to recognize that it has been disconnected.

> **Note**
>
> By forcing a disconnect, a client has to re-authenticate and re-authorize with a fresh session state. The API call itself doesn't prevent reconnection of clients. If this is desired, then additionally a client's credential or policy must be modified before issuing the `DeleteConnection` API call.

For information about pricing, see AWS IoT Core pricing.

Use cases

The `DeleteConnection` API is useful for managing misbehaving clients that exhibit problematic behavior or consume excessive resources. By forcing a disconnect, you can ensure that clients re-establish their connections with proper authentication and authorization, which can help resolve resource consumption problems.

Client redirection scenarios also benefit from this API. When you need to redirect clients to different endpoints or AWS Regions, you can disconnect them programmatically and have them reconnect to a different AWS IoT Core endpoint by changing DNS settings. This API can help resolve stuck connections or clear problematic session states that might be preventing normal operation.

API parameters

The `DeleteConnection` API accepts the following parameters:

clientId (required)

The unique identifier of the MQTT client to disconnect. This is specified in the URL path. The client ID can't start with a dollar sign ($).

> **Note**
>
> MQTT client IDs can contain characters that are not valid in HTTP requests. When using the `DeleteConnection` API, you must URL encode (percent-encode) any characters in

the client ID that are valid in MQTT but not in HTTP. This includes special characters like spaces, forward slashes (/), and UTF-8 characters. For example, a space becomes %20, a forward slash becomes %2F, and the UTF-8 character ü becomes %C3%BC. Proper encoding ensures that MQTT client IDs are correctly transmitted in the HTTP-based API call.

cleanSession (optional)

Specifies whether to remove the client's session state when disconnecting. Set to `true` to delete all session information, including subscriptions and queued messages. Set to `false` to preserve the session state. By default, this is set to `false` (preserves the session state). For clean sessions this parameter will be ignored.

preventWillMessage (optional)

Controls whether AWS IoT Core dispatches the Last Will and Testament (LWT) message if available upon disconnection. Set to `true` to prevent dispatching the LWT message. Set to `false` to allow dispatching. By default, this is set to `false` (dispatches LWT if available).

API syntax

The `DeleteConnection` API uses the following HTTP request format:

```
DELETE /connections/<clientId>?
cleanSession=<cleanSession>&preventWillMessage=<preventWillMessage> HTTP/1.1
```

Example requests:

```
// Basic disconnect (preserves session, allows LWT message)
DELETE /connections/myDevice123 HTTP/1.1

// Disconnect and clear session
DELETE /connections/myDevice123?cleanSession=TRUE HTTP/1.1

// Disconnect, clear session, and prevent LWT message
DELETE /connections/myDevice123?cleanSession=TRUE&preventWillMessage=TRUE HTTP/1.1
```

Successful requests return HTTP 200 OK with no response body.

> **Note**
>
> The service name used by [AWS Signature Version 4](#) to sign requests is: *iotdevicegateway*. You can find your endpoint by using the `aws iot describe-endpoint --endpoint-type iot:Data-ATS` command.

Required permissions

To use the `DeleteConnection` API, you need the following IAM permission:

```
iot:DeleteConnection
```

You can scope this permission to specific client IDs using resource-based policies. For example:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:DeleteConnection",
            "Resource": "arn:aws:iot:region:account:client/myDevice*"
        }
    ]
}
```

Important considerations

Disconnected clients can immediately attempt to reconnect unless you have implemented additional logic to prevent reconnection. The disconnect operation only terminates the current connection and doesn't prevent a connection from reconnecting. If you need to prevent reconnection, consider implementing client-side logic or disabling the credential of a device.

Rate limits apply to the API as part of standard AWS IoT Core API rate limiting. When planning bulk disconnection operations, ensure you account for these limits and implement appropriate retry logic and batching strategies to avoid throttling. For more information, see [AWS IoT Core endpoints and quotas](#).

Error responses

The `DeleteConnection` API can return the following error responses:

InvalidRequestException

The request is not valid. This can occur if the client ID format is invalid, contains a dollar sign ($) prefix, or if required parameters are missing.

ResourceNotFoundException

The specified client ID does not exist or is not currently connected, and doesn't have a persistent session.

UnauthorizedException

You are not authorized to perform this operation. Ensure you have the required `iot:DeleteConnection` permission.

ForbiddenException

The caller isn't authorized to make the request. This might occur due to insufficient IAM permissions or resource-based policy restrictions.

ThrottlingException

The rate exceeds the limit. Reduce the frequency of your API calls and implement appropriate retry logic with exponential backoff.

InternalFailureException

An unexpected error has occurred. Retry the request after a brief delay.

ServiceUnavailableException

The service is temporarily unavailable. Retry the request after a brief delay.

HTTPS publish

Clients can publish messages by making requests to the REST API using the HTTP 1.0 or 1.1 protocols. For the authentication and port mappings used by HTTP requests, see ???.

> **Note**
>
> HTTPS doesn't support a `clientId` value like MQTT does. `clientId` is available when using MQTT, but it's not available when using HTTPS.

HTTPS message URL

Devices and clients publish their messages by making POST requests to a client-specific endpoint and a topic-specific URL:

```
https://IoT_data_endpoint/topics/url_encoded_topic_name?qos=1
```

- *IoT_data_endpoint* is the AWS IoT device data endpoint. You can find the endpoint in the AWS IoT console on the thing's details page or on the client by using the AWS CLI command:

  ```
  aws iot describe-endpoint --endpoint-type iot:Data-ATS
  ```

  The endpoint should look something like this: `a3qjEXAMPLEffp-ats.iot.us-west-2.amazonaws.com`
- *url_encoded_topic_name* is the full topic name of the message being sent.

HTTPS message code examples

These are some examples of how to send an HTTPS message to AWS IoT.

Python (port 8443)

```
import requests
import argparse

# define command-line parameters
parser = argparse.ArgumentParser(description="Send messages through an HTTPS
 connection.")
parser.add_argument('--endpoint', required=True, help="Your AWS IoT data custom
 endpoint, not including a port. " +
                                                      "Ex: \"abcdEXAMPLExyz-
ats.iot.us-east-1.amazonaws.com\"")
parser.add_argument('--cert', required=True, help="File path to your client
 certificate, in PEM format.")
parser.add_argument('--key', required=True, help="File path to your private key, in
 PEM format.")
parser.add_argument('--topic', required=True, default="test/topic", help="Topic to
 publish messages to.")
parser.add_argument('--message', default="Hello World!", help="Message to publish. "
 +
```

```
                                                                "Specify empty string to
 publish nothing.")

# parse and load command-line parameter values
args = parser.parse_args()

# create and format values for HTTPS request
publish_url = 'https://' + args.endpoint + ':8443/topics/' + args.topic + '?qos=1'
publish_msg = args.message.encode('utf-8')

# make request
publish = requests.request('POST',
            publish_url,
            data=publish_msg,
            cert=[args.cert, args.key])

# print results
print("Response status: ", str(publish.status_code))
if publish.status_code == 200:
        print("Response body:", publish.text)
```

Python (port 443)

```
import requests
import http.client
import json
import ssl

ssl_context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_CLIENT)
ssl_context.minimum_version = ssl.TLSVersion.TLSv1_2

# note the use of ALPN
ssl_context.set_alpn_protocols(["x-amzn-http-ca"])
ssl_context.load_verify_locations(cafile="./<root_certificate>")

# update the certificate and the AWS endpoint
ssl_context.load_cert_chain("./<certificate_in_PEM_Format>",
 "<private_key_in_PEM_format>")
connection = http.client.HTTPSConnection('<the ats IoT endpoint>', 443,
 context=ssl_context)
message = {'data': 'Hello, I'm using TLS Client authentication!'}
json_data = json.dumps(message)
connection.request('POST', '/topics/device%2Fmessage?qos=1', json_data)
```

```
# make request
response = connection.getresponse()

# print results
print(response.read().decode())
```

CURL

You can use curl from a client or device to send a message to AWS IoT.

To use curl to send a message from an AWS IoT client device

1. Check the **curl** version.

   a. On your client, run this command at a command prompt.

      curl --help

      In the help text, look for the TLS options. You should see the `--tlsv1.2` option.

   b. If you see the `--tlsv1.2` option, continue.

   c. If you don't see the `--tlsv1.2` option or you get a `command not found` error, you might need to update or install curl on your client or install `openssl` before you continue.

2. Install the certificates on your client.

   Copy the certificate files that you created when you registered your client (thing) in the AWS IoT console. Make sure you have these three certificate files on your client before you continue.

   - The CA certificate file (*Amazon-root-CA-1.pem* in this example).
   - The client's certificate file (*device.pem.crt* in this example).
   - The client's private key file (*private.pem.key* in this example).

3. Create the **curl** command line, replacing the replaceable values for those of your account and system.

```
curl --tlsv1.2 \
    --cacert Amazon-root-CA-1.pem \
    --cert device.pem.crt \
    --key private.pem.key \
```

```
    --request POST \
    --data "{ \"message\": \"Hello, world\" }" \
    "https://IoT_data_endpoint:8443/topics/topic?qos=1"
```

--tlsv1.2

Use TLS 1.2 (SSL).

--cacert *Amazon-root-CA-1.pem*

The file name and path, if necessary, of the CA certificate to verify the peer.

--cert *device.pem.crt*

The client's certificate file name and path, if necessary.

--key *private.pem.key*

The client's private key file name and path, if necessary.

--request POST

The type of HTTP request (in this case, POST).

--data "*{ \"message\": \"Hello, world\" }*"

The HTTP POST data you want to publish. In this case, it's a JSON string, with the internal quotation marks escaped with the backslash character (\).

"https://*IoT_data_endpoint*:8443/topics/*topic*?qos=1"

The URL of your client's AWS IoT device data endpoint, followed by the HTTPS port, `:8443`, which is then followed by the keyword, `/topics/` and the topic name, `topic`, in this case. Specify the Quality of Service as the query parameter, `?qos=1`.

4. Open the MQTT test client in the AWS IoT console.

   Follow the instructions in View MQTT messages with the AWS IoT MQTT client and configure the console to subscribe to messages with the topic name of *topic* used in your **curl** command, or use the wildcard topic filter of `#`.

5. Test the command.

   While monitoring the topic in the test client of the AWS IoT console, go to your client and issue the curl command line that you created in step 3. You should see your client's messages in the console.

MQTT topics

MQTT topics identify AWS IoT messages. AWS IoT clients identify the messages they publish by giving the messages topic names. Clients identify the messages to which they want to subscribe (receive) by registering a topic filter with AWS IoT Core. The message broker uses topic names and topic filters to route messages from publishing clients to subscribing clients.

The message broker uses topics to identify messages sent using MQTT and sent using HTTP to the HTTPS message URL.

While AWS IoT supports some reserved system topics, most MQTT topics are created and managed by you, the system designer. AWS IoT uses topics to identify messages received from publishing clients and select messages to send to subscribing clients, as described in the following sections. Before you create a topic namespace for your system, review the characteristics of MQTT topics to create the hierarchy of topic names that works best for your IoT system.

Topic names

Topic names and topic filters are UTF-8 encoded strings. They can represent a hierarchy of information by using the forward slash (/) character to separate the levels of the hierarchy. For example, this topic name could refer to a temperature sensor in room 1:

- `sensor/temperature/room1`

In this example, there might also be other types of sensors in other rooms with topic names such as:

- `sensor/temperature/room2`
- `sensor/humidity/room1`
- `sensor/humidity/room2`

> **Note**
>
> As you consider topic names for the messages in your system, keep in mind:
>
> - Topic names and topic filters are case sensitive.
> - Topic names must not contain personally identifiable information.

- Topic names that begin with a $ are reserved topics to be used only by AWS IoT Core.
- AWS IoT Core can't send or receive messages between AWS accounts or Regions.

For more information on designing your topic names and namespace, see our whitepaper, Designing MQTT Topics for AWS IoT Core.

For examples of how apps can publish and subscribe to messages, start with Getting started with AWS IoT Core tutorials and AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client.

> **Important**
>
> The topic namespace is limited to an AWS account and Region. For example, the `sensor/temp/room1` topic used by an AWS account in one Region is distinct from the `sensor/temp/room1` topic used by the same AWS account in another Region or used by any other AWS account in any Region.

Topic ARN

All topic ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topic/Topic
```

For example, `arn:aws:iot:us-west-2:123EXAMPLE456:topic/application/topic/device/sensor` is an ARN for the topic `application/topic/device/sensor`.

Topic name filters

Subscribing clients register topic name filters with the message broker to specify the message topics that the message broker should send to them. A topic name filter can be a single topic name to subscribe to a single topic name or it can include wildcard characters to subscribe to multiple topic names at the same time.

Publishing clients can't use wildcard characters in the topic names they publish.

The following table lists the wildcard characters that can be used in a topic filter.

Topic wildcards

Wildcard character	Matches	Notes
#	All strings at and below its level in the topic hierarchy.	Must be the last character in the topic filter. Must be the only character in its level of the topic hierarchy. Can be used in a topic filter that also contains the + wildcard character.
+	Any string in the level that contains the character.	Must be the only character in its level of the topic hierarchy. Can be used in multiple levels of a topic filter.

Using wildcards with the previous sensor topic name examples:

- A subscription to `sensor/#` receives messages published to `sensor/`, `sensor/temperature`, `sensor/temperature/room1`, but not messages published to `sensor`.
- A subscription to `sensor/+/room1` receives messages published to `sensor/temperature/room1` and `sensor/humidity/room1`, but not messages sent to `sensor/temperature/room2` or `sensor/humidity/room2`.

Topic filter ARN

All topic filter ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topicfilter/TopicFilter
```

For example, `arn:aws:iot:us-west-2:123EXAMPLE456:topicfilter/application/topic/+/sensor` is an ARN for the topic filter `application/topic/+/sensor`.

MQTT message payload

The message payload that is sent in your MQTT messages isn't specified by AWS IoT, unless it's for one of the the section called "Reserved topics". To accommodate your application's needs, we recommend you define the message payload for your topics within the constraints of the AWS IoT Core Service Quotas for Protocols.

Using a JSON format for your message payload enables the AWS IoT rules engine to parse your messages and apply SQL queries to it. If your application doesn't require the rules engine to apply SQL queries to your message payloads, you can use any data format that your application requires. For information about limitations and reserved characters in a JSON document used in SQL queries, see JSON extensions.

For more information about designing your MQTT topics and their corresponding message payloads, see Designing MQTT Topics for AWS IoT Core.

If a message size limit exceeds the service quotas, it will result in a `CLIENT_ERROR` with reason `PAYLOAD_LIMIT_EXCEEDED` and "Message payload exceeds size limit for message type." For more information about message size limit, see AWS IoT Core message broker limits and quotas.

Reserved topics

Topics that begin with a dollar sign ($) are reserved for use by AWS IoT. You can subscribe and publish to these reserved topics as they allow; however, you can't create new topics that begin with a dollar sign. Unsupported publish or subscribe operations to reserved topics can result in a terminated connection.

Asset model topics

Topic	Client operations allowed	Description
$aws/sitewise/asset-models/*assetModelId* /assets/*assetId*/properties/*propertyId*	Subscribe	AWS IoT SiteWise publishes asset property notifications to this topic. For more information, see Interacting with other AWS services in the *AWS IoT SiteWise User Guide*.

AWS IoT Device Defender topics

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the *payload-format* of the topic. AWS IoT Device Defender topics only support MQTT publish.

payload-format	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

For more information, see Sending metrics from devices.

Topic	Allowed operations	Description
$aws/things/*thingName* /defender/metrics/ *payload-format*	Publish	AWS IoT Device Defender agents publish metrics to this topic. For more informati on, see Sending metrics from devices.
$aws/things/*thingName* /defender/metrics/ *payload-format* / accepted	Subscribe	AWS IoT publishes to this topic after a AWS IoT Device Defender agent publishes a successful message to $aws/ things/*thingName* /defender/metrics/ *payload-format* . For more informati on, see Sending metrics from devices.
$aws/things/*thingName* /defender/metrics/ *payload-format* / rejected	Subscribe	AWS IoT publishes to this topic after a AWS IoT Device Defender agent publishes an unsuccessful message to $aws/things/*thingName* /defender/ metrics/*payload-format* . For more information, see Sending metrics from devices.

AWS IoT Core Device Location topics

AWS IoT Core Device Location can resolve the measurement data from your device and provide an estimated location of your IoT devices. The measurement data from the device can include GNSS, Wi-Fi, cellular, and IP address. AWS IoT Core Device Location then chooses the measurement type that provides the best accuracy and solves the device location information. For more information, see AWS IoT Core Device Location and Resolving device location using AWS IoT Core Device Location MQTT topics.

Topic	Allowed operations	Description
$aws/device_location/*customer_device_id*/get_position_estimate	Publish	A device publishes to this topic to get the scanned raw measurement data to be resolved by AWS IoT Core Device Location.
$aws/device_location/*customer_device_id*/get_position_estimate/accepted	Subscribe	AWS IoT Core Device Location publishes to this topic after it has resolved the device location successfully.
$aws/device_location/*customer_device_id*/get_position_estimate/rejected	Subscribe	AWS IoT Core Device Location publishes to this topic when it is unable to resolve the device location successfully due to 4xx errors.

Event topics

The event messages are published when certain events occur. For example, events are generated by the registry when things are added, updated, or deleted. The table shows the various AWS IoT events and their reserved topics.

Topic	Client operations allowed	Description
$aws/events/certificates/registered/*caCertificateId*	Subscribe	AWS IoT publishes this message when AWS IoT automatically registers a certificate and when a client presents

Topic	Client operations allowed	Description
		a certificate with the PENDING_A CTIVATION status. For more informati on, see the section called "Configure the first connection by a client for automatic registration".
$aws/events/job/*jobID*/canceled	Subscribe	AWS IoT publishes this message when a job is canceled. For more information, see Jobs events.
$aws/events/job/*jobID*/cancellation_in_progress	Subscribe	AWS IoT publishes this message when a job cancellation is in progress. For more information, see Jobs events.
$aws/events/job/*jobID*/completed	Subscribe	AWS IoT publishes this message when a job has completed. For more informati on, see Jobs events.
$aws/events/job/*jobID*/deleted	Subscribe	AWS IoT publishes this message when a job is deleted. For more information, see Jobs events.
$aws/events/job/*jobID*/deletion_in_progress	Subscribe	AWS IoT publishes this message when a job deletion is in progress. For more information, see Jobs events.
$aws/events/jobExe cution/*jobID*/canceled	Subscribe	AWS IoT publishes this message when a job execution is canceled. For more information, see Jobs events.
$aws/events/jobExe cution/*jobID*/deleted	Subscribe	AWS IoT publishes this message when a job execution is deleted. For more information, see Jobs events.

Topic	Client operations allowed	Description
$aws/events/jobExecution/*jobID*/failed	Subscribe	AWS IoT publishes this message when a job execution has failed. For more information, see Jobs events.
$aws/events/jobExecution/*jobID*/rejected	Subscribe	AWS IoT publishes this message when a job execution was rejected. For more information, see Jobs events.
$aws/events/jobExecution/*jobID*/removed	Subscribe	AWS IoT publishes this message when a job execution was removed. For more information, see Jobs events.
$aws/events/jobExecution/*jobID*/succeeded	Subscribe	AWS IoT publishes this message when a job execution succeeded. For more information, see Jobs events.
$aws/events/jobExecution/*jobID*/timed_out	Subscribe	AWS IoT publishes this message when a job execution timed out. For more information, see Jobs events.
$aws/events/presence/connected/*clientId*	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT. For more information, see Connect/Disconnect events.
$aws/events/presence/disconnected/*clientId*	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID disconnects to AWS IoT. For more information, see Connect/Disconnect events.

Topic	Client operations allowed	Description
$aws/events/subscriptions/subscribed/*clientId*	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID subscribes to an MQTT topic. For more information, see Subscribe/Unsubscribe events.
$aws/events/subscriptions/unsubscribed/*clientId*	Subscribe	AWS IoT publishes to this topic when an MQTT client with the specified client ID unsubscribes to an MQTT topic. For more information, see Subscribe/Unsubscribe events.
$aws/events/thing/*thingName*/created	Subscribe	AWS IoT publishes to this topic when the *thingName* thing is created. For more information, see the section called "Registry events".
$aws/events/thing/*thingName*/updated	Subscribe	AWS IoT publishes to this topic when the *thingName* thing is updated. For more information, see the section called "Registry events".
$aws/events/thing/*thingName*/deleted	Subscribe	AWS IoT publishes to this topic when the *thingName* thing is deleted. For more information, see the section called "Registry events".
$aws/events/thingGroup/*thingGroupName*/created	Subscribe	AWS IoT publishes to this topic when thing group *thingGroupName* is created. For more information, see the section called "Registry events".

Topic	Client operations allowed	Description
$aws/events/thingGroup/*thingGroupName* /updated	Subscribe	AWS IoT publishes to this topic when thing group *thingGroupName* is updated. For more information, see the section called "Registry events".
$aws/events/thingGroup/*thingGroupName* /deleted	Subscribe	AWS IoT publishes to this topic when thing group *thingGroupName* is deleted. For more information, see the section called "Registry events".
$aws/events/thingType/*thingTypeName* /created	Subscribe	AWS IoT publishes to this topic when the *thingTypeName* thing type is created. For more information, see the section called "Registry events".
$aws/events/thingType/*thingTypeName* /updated	Subscribe	AWS IoT publishes to this topic when the *thingTypeName* thing type is updated. For more information, see the section called "Registry events".
$aws/events/thingType/*thingTypeName* /deleted	Subscribe	AWS IoT publishes to this topic when the *thingTypeName* thing type is deleted. For more information, see the section called "Registry events".
$aws/events/thingTypeAssociation/thing/*thingName* /*thingTypeName*	Subscribe	AWS IoT publishes to this topic when thing *thingName* is associated with or disassociated from thing type *thingTypeName* . For more information, see the section called "Registry events".

Topic	Client operations allowed	Description
$aws/events/thingGroupMembership/thingGroup/*thingGroupName*/thing/*thingName*/added	Subscribe	AWS IoT publishes to this topic when thing *thingName* is added to thing group *thingGroupName*. For more information, see the section called "Registry events".
$aws/events/thingGroupMembership/thingGroup/*thingGroupName*/thing/*thingName*/removed	Subscribe	AWS IoT publishes to this topic when thing *thingName* is removed from thing group *thingGroupName*. For more information, see the section called "Registry events".
$aws/events/thingGroupHierarchy/thingGroup/*parentThingGroupName*/childThingGroup/*childThingGroupName*/added	Subscribe	AWS IoT publishes to this topic when thing group *childThingGroupName* is added to thing group *parentThingGroupName*. For more information, see the section called "Registry events".
$aws/events/thingGroupHierarchy/thingGroup/*parentThingGroupName*/childThingGroup/*childThingGroupName*/removed	Subscribe	AWS IoT publishes to this topic when thing group *childThingGroupName* is removed from thing group *parentThingGroupName*. For more information, see the section called "Registry events".

Fleet provisioning topics

> **Note**
>
> The client operations noted as **Receive** in this table indicate topics that AWS IoT publishes directly to the client that requested it, whether the client has subscribed to the topic or not. Clients should expect to receive these response messages even if they haven't subscribed to

them. These response messages don't pass through the message broker and they can't be subscribed to by other clients or rules.

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the *payload-format* of the topic.

payload-format	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

For more information, see Device provisioning MQTT API.

Topic	Client operations allowed	Description
$aws/certificates/create/*payload-format*	Publish	Publish to this topic to create a certificate from a certificate signing request (CSR).
$aws/certificates/create/*payload-format*/accepted	Subscribe, Receive	AWS IoT publishes to this topic after a successful call to $aws/certificates/create/*payload-format*.
$aws/certificates/create/*payload-format*/rejected	Subscribe, Receive	AWS IoT publishes to this topic after an unsuccessful call to $aws/certificates/create/*payload-format*.
$aws/certificates/create-from-csr/*payload-format*	Publish	Publishes to this topic to create a certificate from a CSR.
$aws/certificates/create-from-csr/*payload-format*/accepted	Subscribe, Receive	AWS IoT publishes to this topic a successful call to $aws/certificates/create-from-csr/*payload-format*.

Topic	Client operations allowed	Description
$aws/certificates/create-from-csr/*payload-format* /rejected	Subscribe, Receive	AWS IoT publishes to this topic an unsuccessful call to $aws/certificates/create-from-csr/*payload-format* .
$aws/provisioning-templates/*templateName* /provision/*payload-format*	Publish	Publish to this topic to register a thing.
$aws/provisioning-templates/*templateName* /provision/*payload-format* /accepted	Subscribe, Receive	AWS IoT publishes to this topic after a successful call to $aws/provisioning-templates/*templateName* /provision/*payload-format* .
$aws/provisioning-templates/*templateName* /provision/*payload-format* /rejected	Subscribe, Receive	AWS IoT publishes to this topic after an unsuccessful call to $aws/provisioning-templates/*templateName* /provision/*payload-format* .

Job topics

> **Note**
>
> The client operations noted as **Receive** in this table indicate topics that AWS IoT publishes directly to the client that requested it, whether the client has subscribed to the topic or not. Clients should expect to receive these response messages even if they haven't subscribed to them.
>
> These response messages don't pass through the message broker and they can't be subscribed to by other clients or rules. To subscribe to job activity related messages, use the `notify` and `notify-next` topics.
>
> When subscribing to the job and `jobExecution` event topics for your fleet-monitoring solution, you must first enable [job and job execution events](#) to receive any events on the cloud side.

For more information, see Jobs device MQTT API operations.

Topic	Client operations allowed	Description
$aws/things/*thingName*/jobs/get	Publish	Devices publish a message to this topic to make a `GetPendingJobExecutions` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/get/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses from a `GetPendingJobExecutions` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/get/rejected	Subscribe, Receive	Devices subscribe to this topic to receive a response when a `GetPendingJobExecutions` request is rejected. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/start-next	Publish	Devices publish a message to this topic to make a `StartNextPendingJobExecution` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/start-next/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to a `StartNextPendingJobExecution` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/start-next/rejected	Subscribe, Receive	Devices subscribe to this topic to receive a response when a `StartNextPendingJobExecution` request

Topic	Client operations allowed	Description
		is rejected. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/*jobId*/get	Publish	Devices publish a message to this topic to make a `DescribeJobExecution` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/*jobId*/get/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to a `DescribeJobExecution` request. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/*jobId*/get/rejected	Subscribe, Receive	Devices subscribe to this topic to receive a response when a `DescribeJobExecution` request is rejected. For more information, see Jobs device MQTT API operations.
$aws/things/*thingName*/jobs/*jobId*/update	Publish	Devices publish a message to this topic to make an `UpdateJobExecution` request. For more information, see Jobs device MQTT API operations.

Topic	Client operations allowed	Description
$aws/things/*thingName*/jobs/*jobId*/update/accepted	Subscribe, Receive	Devices subscribe to this topic to receive successful responses to an `UpdateJobExecution` request. For more information, see [Jobs device MQTT API operations](). **Note** Only the device that publishes to $aws/things/*thingName*/jobs/*jobId*/update receives messages on this topic.
$aws/things/*thingName*/jobs/*jobId*/update/rejected	Subscribe, Receive	Devices subscribe to this topic to receive a response when an `UpdateJobExecution` request is rejected. For more information, see [Jobs device MQTT API operations](). **Note** Only the device that publishes to $aws/things/*thingName*/jobs/*jobId*/update receives messages on this topic.
$aws/things/*thingName*/jobs/notify	Subscribe, Receive	Devices subscribe to this topic to receive notifications when a job execution is added or removed to the list of pending executions for a thing. For more information, see [Jobs device MQTT API operations]().

Topic	Client operations allowed	Description
$aws/things/*thingName*/jobs/notify-next	Subscribe, Receive	Devices subscribe to this topic to receive notifications when the next pending job execution for the thing is changed. For more information, see Jobs device MQTT API operations.
$aws/events/job/*jobId*/completed	Subscribe	The Jobs service publishes an event on this topic when a job completes. For more information, see Jobs events.
$aws/events/job/*jobId*/canceled	Subscribe	The Jobs service publishes an event on this topic when a job is canceled. For more information, see Jobs events.
$aws/events/job/*jobId*/deleted	Subscribe	The Jobs service publishes an event on this topic when a job is deleted. For more information, see Jobs events.
$aws/events/job/*jobId*/cancellation_in_progress	Subscribe	The Jobs service publishes an event on this topic when a job cancellation begins. For more information, see Jobs events.
$aws/events/job/*jobId*/deletion_in_progress	Subscribe	The Jobs service publishes an event on this topic when a job deletion begins. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/succeeded	Subscribe	The Jobs service publishes an event on this topic when job execution succeeds. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/failed	Subscribe	The Jobs service publishes an event on this topic when a job execution fails. For more information, see Jobs events.

Topic	Client operations allowed	Description
$aws/events/jobExecution/*jobId*/rejected	Subscribe	The Jobs service publishes an event on this topic when a job execution is rejected. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/canceled	Subscribe	The Jobs service publishes an event on this topic when a job execution is canceled. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/timed_out	Subscribe	The Jobs service publishes an event on this topic when a job execution times out. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/removed	Subscribe	The Jobs service publishes an event on this topic when a job execution is removed. For more information, see Jobs events.
$aws/events/jobExecution/*jobId*/deleted	Subscribe	The Jobs service publishes an event on this topic when a job execution is deleted. For more information, see Jobs events.

Commands topics

> **Note**
>
> The client operations noted as **Receive** in this table indicate topics that AWS IoT publishes directly to the client that requested it, whether the client has subscribed to the topic or not. Clients should expect to receive these response messages even if they haven't subscribed to them.

These response messages don't pass through the message broker and they can't be subscribed to by other clients or rules.

Topic	Client operations allowed	Description
$aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / request/*<PayloadFormat>* $aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / request	Subscribe, Receive	Devices receive a message on this topic when a request is made to start a command execution from the console or using the `StartCommandExecution` API. In this case, the *<devices>* can be either IoT things or MQTT clients, and *<DeviceID>* can be either the IoT thing name or the MQTT client ID.
$aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / response/*<PayloadFormat>*	Publish	Devices use the `UpdateCommandExecution` MQTT API to publish a message to this topic about the command execution. The message is published as a response to the request to start a command execution from the console or using the `StartCommandExecution` API. The published message will use JSON or CBOR as the *<PayloadFormat>* .
$aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / response/accepted /*<PayloadFormat>*	Subscribe, Receive	If the cloud service successfully processed the command execution result, AWS IoT Device Management publishes a response to the /accepted topic.

Topic	Client operations allowed	Description
$aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / response/accepted		
$aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / response/rejected /*<PayloadFormat>* $aws/commands/*<devices>* /*<DeviceID>* /executions/*<ExecutionId>* / response/rejected	Publish	If the cloud service failed to process the command execution result, AWS IoT Device Management publishes a response to the /rejected topic.

Rule topics

Topic	Client operations allowed	Description
$aws/rules/*ruleName*	Publish	A device or an application publishes to this topic to trigger rules directly. For more information, see Reducing messaging costs with Basic Ingest.

Secure tunneling topics

Topic	Client operations allowed	Description
$aws/things/*thing-name*/tunnels/notify	Subscribe	AWS IoT publishes this message for an IoT agent to start a local proxy on the remote device. For more information, see [the section called "IoT agent snippet"]().

Shadow topics

The topics in this section are used by named and unnamed shadows. The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

ShadowTopicPrefix value	Shadow type
$aws/things/*thingName*/shadow	Unnamed (classic) shadow
$aws/things/*thingName*/shadow/name/*shadowName*	Named shadow

To create a complete topic, select the *ShadowTopicPrefix* for the type of shadow to which you want to refer, replace *thingName* and if applicable, *shadowName*, with their corresponding values, and then append that with the topic stub as shown in the following table. Remember that topics are case sensitive.

Topic	Client operations allowed	Description
ShadowTopicPrefix/delete	Publish/Subscribe	A device or an application publishes to this topic to delete a shadow. For more information, see [/delete]().
ShadowTopicPrefix/delete/accepted	Subscribe	The Device Shadow service sends messages to this topic when a shadow

Topic	Client operations allowed	Description
		is deleted. For more information, see /delete/accepted.
ShadowTopicPrefix / delete/rejected	Subscribe	The Device Shadow service sends messages to this topic when a request to delete a shadow is rejected. For more information, see /delete/rejected.
ShadowTopicPrefix / get	Publish/Subscribe	An application or a thing publishes an empty message to this topic to get a shadow. For more information, see Device Shadow MQTT topics.
ShadowTopicPrefix / get/accepted	Subscribe	The Device Shadow service sends messages to this topic when a request for a shadow is made successfully. For more information, see /get/accepted.
ShadowTopicPrefix / get/rejected	Subscribe	The Device Shadow service sends messages to this topic when a request for a shadow is rejected. For more information, see /get/rejected.
ShadowTopicPrefix / update	Publish/Subscribe	A thing or application publishes to this topic to update a shadow. For more information, see /update.
ShadowTopicPrefix / update/accepted	Subscribe	The Device Shadow service sends messages to this topic when an update is successfully made to a shadow. For more information, see /update/accepted.

Topic	Client operations allowed	Description
ShadowTopicPrefix / update/rejected	Subscribe	The Device Shadow service sends messages to this topic when an update to a shadow is rejected. For more information, see [/update/rejected]().
ShadowTopicPrefix / update/delta	Subscribe	The Device Shadow service sends messages to this topic when a difference is detected between the reported and desired sections of a shadow. For more information, see [/update/delta]().
ShadowTopicPrefix / update/documents	Subscribe	AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed. For more information, see [/update/documents]().

MQTT-based file delivery topics

> **Note**
>
> The client operations noted as **Receive** in this table indicate topics that AWS IoT publishes directly to the client that requested it, whether the client has subscribed to the topic or not. Clients should expect to receive these response messages even if they haven't subscribed to them. These response messages don't pass through the message broker and they can't be subscribed to by other clients or rules.

These messages support response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the *payload-format* of the topic.

payload-format	**Response format data type**
cbor	Concise Binary Object Representation (CBOR)

payload-format	Response format data type
json	JavaScript Object Notation (JSON)

Topic	Client operations allowed	Description
$aws/things/*ThingName*/streams/*StreamId*/data/*payload-format*	Subscribe, Receive	AWS MQTT-based file delivery publishes to this topic if the "GetStream" request from a device is accepted. The payload contains the stream data. For more information, see Using AWS IoT MQTT-based file delivery in devices.
$aws/things/*ThingName*/streams/*StreamId*/get/*payload-format*	Publish	A device publishes to this topic to perform a "GetStream" request. For more information, see Using AWS IoT MQTT-based file delivery in devices.
$aws/things/*ThingName*/streams/*StreamId*/description/*payload-format*	Subscribe, Receive	AWS MQTT-based file delivery publishes to this topic if the "DescribeStream" request from a device is accepted. The payload contains the stream description. For more information, see Using AWS IoT MQTT-based file delivery in devices.
$aws/things/*ThingName*/streams/*StreamId*/describe/*payload-format*	Publish	A device publishes to this topic to perform a "DescribeStream" request. For more information, see Using AWS IoT MQTT-based file delivery in devices.
$aws/things/*ThingName*/streams/*StreamId*/	Subscribe, Receive	AWS MQTT-based file delivery publishes to this topic if a "Describe Stream" or "GetStream" request

Topic	Client operations allowed	Description
rejected/*payload-format*		from a device is rejected. For more information, see Using AWS IoT MQTT-based file delivery in devices.

Reserved topic ARN

All reserved topic ARNs (Amazon Resource Names) have the following form:

```
arn:aws:iot:aws-region:AWS-account-ID:topic/Topic
```

For example, `arn:aws:iot:us-west-2:123EXAMPLE456:topic/$aws/things/thingName/jobs/get/accepted` is an ARN for the reserved topic `$aws/things/thingName/jobs/get/accepted`.

Domain configurations

In AWS IoT Core, you can use domain configurations to configure and manage the behaviors of your data endpoints. With domain configurations, you can generate multiple AWS IoT Core data endpoints, customize them with your own fully qualified domain names (FQDN) and associated server certificates, and also associate a custom authorizer. For more information, see Custom authentication and authorization.

> **Note**
>
> This feature is not available in AWS GovCloud (US) AWS Regions.

In this chapter:

- What is a domain configuration?
- Creating and configuring AWS managed domains
- Creating and configuring customer managed domains
- Managing domain configurations
- Configuring TLS settings in domain configurations

- Server certificate configuration for OCSP stapling

What is a domain configuration?

In AWS IoT Core, a domain configuration refers to the setup and configuration of a domain (either AWS managed domain or customer managed domain) for your AWS IoT Core data endpoints. AWS IoT Core also provides a default endpoint for your AWS account (`iot:Data-ATS`) for devices to communicate with AWS IoT Core.

In this topic:

- Use cases
- Key concepts
- Important notes

Use cases

You can use domain configurations to simplify tasks like the following.

- Migrate devices to AWS IoT Core.
- Support heterogeneous device fleets by maintaining separate domain configurations for separate device types.
- Maintain brand identity (for example, through domain name) while migrating application infrastructure to AWS IoT Core.

Key concepts

The following concepts provide details about domain configurations and related concepts.

- **Domain configuration**

  The setup and configuration of a domain for your AWS IoT Core endpoints.

- **Default endpoint domain**

  The domain that AWS IoT provides with the default endpoint such as `iot:Data-ATS`. To find the default endpoint, run the describe-endpoint or describe-domain-configuration CLI command.

Alternatively, go to AWS IoT Core console, choose **Domain configurations** from **Connect** on the left navigation. The default endpoint is listed with the name `iot:Data-ATS`.

- **AWS managed domain**

  The domain that AWS will manage. Choosing AWS managed domain means that your devices will connect using a data endpoint provided by AWS. AWS will manage the domain and the certificates.

- **Customer managed domain**

  The domain that you will manage. Also known as custom domain. Choosing customer managed domain means that your devices will connect using a custom domain data endpoint. You will manage the domain and the certificates. Customer managed domain allows you to tailor the endpoint URLs to suit your needs. For example, you can use a custom domain name (`your-domain-name.com`) or apply specific access policies.

- **Authentication type**

  The authentication type that you choose to authenticate your devices when connecting to AWS IoT Core. When creating a domain configuration, you must specify an authentication type. For more information, see ???.

- **Application protocol**

  The application layer protocols which your devices use when connecting to AWS IoT Core. When creating a domain configuration, you must specify an application protocol. For more information, see ???.

Important notes

AWS IoT Core uses the server name indication (SNI) TLS extension to apply domain configurations. When connecting devices to AWS IoT Core, clients can send the Server Name Indication (SNI) extension, which is required for features such as multi-account registration, configurable endpoints, custom domains, and VPC endpoints. They also must pass a server name that is identical to the domain name that you specify in the domain configuration. To test this service, use the v2 version of the AWS IoT Device SDKs in GitHub.

If you create multiple data endpoints in your AWS account, they will share AWS IoT Core resources such as MQTT topics, device shadows, and rules.

When you provide the server certificates for AWS IoT Core custom domain configuration, the certificates have a maximum of four domain names. For more information, see AWS IoT Core endpoints and quotas.

Creating and configuring AWS managed domains

You create a configurable endpoint on an AWS managed domain by using the CreateDomainConfiguration API. A domain configuration for an AWS managed domain consists of the following:

- `domainConfigurationName`

  A user-defined name that identifies the domain configuration and the value must be unique to your AWS Region. You can't use domain configuration names that start with `IoT:` because they are reserved for default endpoints.
- `defaultAuthorizerName` (optional)

  The name of the custom authorizer to use on the endpoint.
- `allowAuthorizerOverride` (optional)

  A Boolean value that specifies whether devices can override the default authorizer by specifying a different authorizer in the HTTP header of the request. This value is required if a value for `defaultAuthorizerName` is specified.
- `serviceType` (optional)

  The service type that the endpoint delivers. AWS IoT Core only supports the `DATA` service type. When you specify `DATA`, AWS IoT Core returns an endpoint with an endpoint type of `iot:Data-ATS`. You can't create a configurable `iot:Data` (VeriSign) endpoint.
- `TlsConfig` (optional)

  An object that specifies the TLS configuration for a domain. For more information, see ???.

The following example AWS CLI command creates a domain configuration for a `Data` endpoint.

```
aws iot create-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" --service-type "DATA"
```

The output of the command can look like the following.

```
{
    "domainConfigurationName": "myDomainConfigurationName",
    "domainConfigurationArn": "arn:aws:iot:us-east-1:123456789012:domainconfiguration/
myDomainConfigurationName/itihw"
}
```

Creating and configuring customer managed domains

Domain configurations let you specify a custom fully qualified domain name (FQDN) to connect to AWS IoT Core. There are many benefits of using customer managed domains (also known as custom domains): you can expose your own domain or your company's own domain to customers for branding purposes; you can easily change your own domain to point to a new broker; you can support multi-tenancy to serve customers with different domains within the same AWS account; and you can manage your own server certificates details, such as the root certificate authority (CA) used to sign the certificate, the signature algorithm, the certificate chain depth, and the lifecycle of the certificate.

The workflow to set up a domain configuration with a custom domain consists of the following three stages.

1. Registering Server Certificates in AWS Certificate Manager
2. Creating a Domain Configuration
3. Creating DNS Records

Registering server certificates in AWS certificate manager

Before you create a domain configuration with a custom domain, you must register your server certificate chain in AWS Certificate Manager (ACM). You can use the following three types of server certificates.

- ACM Generated Public Certificates
- External Certificates Signed by a Public CA

- External Certificates Signed by a Private CA

> **Note**
>
> AWS IoT Core considers a certificate to be signed by a public CA if it's included in Mozilla's trusted ca-bundle.

Certificate requirements

See Prerequisites for Importing Certificates for the requirements for importing certificates into ACM. In addition to these requirements, AWS IoT Core adds the following requirements.

- The leaf certificate must include the **Extended Key Usage** x509 v3 extension with a value of **serverAuth** (TLS Web Server Authentication). If you request the certificate from ACM, this extension is automatically added.
- The maximum certificate chain depth is 5 certificates.
- The maximum certificate chain size is 16KB.
- The cryptographic algorithms and key sizes that are supported include RSA 2048 bit (RSA_2048) and ECDSA 256 bit (EC_prime256v1).

Using one certificate for multiple domains

If you plan to use one certificate to cover multiple subdomains, use a wildcard domain in the common name (CN) or Subject Alternative Names (SAN) field. For example, use **`*.iot.example.com`** to cover dev.iot.example.com, qa.iot.example.com, and prod.iot.example.com. Each FQDN requires its own domain configuration, but more than one domain configuration can use the same wildcard value. Either the CN or the SAN must cover the FQDN that you want to use as a custom domain. If SANs are present, the CN is ignored and a SAN must cover the FQDN that you want to use as a custom domain. This coverage can be an exact match or a wildcard match. After a wildcard certificate has been validated and registered to an account, other accounts in the region are blocked from creating custom domains that overlap with the certificate.

The following sections describe how to get each type of certificate. Every certificate resource requires an Amazon Resource Name (ARN) registered with ACM that you use when you create your domain configuration.

ACM-generated public certificates

You can generate a public certificate for your custom domain by using the RequestCertificate API. When you generate a certificate in this way, ACM validates your ownership of the custom domain. For more information, see Request a Public Certificate in the *AWS Certificate Manager User Guide*.

External certificates signed by a public CA

If you already have a server certificate that is signed by a public CA (a CA that is included in Mozilla's trusted ca-bundle), you can import the certificate chain directly into ACM by using the ImportCertificate API. To learn more about this task and the prerequisites and certificate format requirements, see Importing Certificates.

External certificates signed by a private CA

If you already have a server certificate that is signed by a private CA or self-signed, you can use the certificate to create your domain configuration, but you also must create an extra public certificate in ACM to validate ownership of your domain. To do this, register your server certificate chain in ACM using the ImportCertificate API. To learn more about this task and the prerequisites and certificate format requirements, see Importing Certificates.

Creating a validation certificate

After you import your certificate to ACM, generate a public certificate for your custom domain by using the RequestCertificate API. When you generate a certificate in this way, ACM validates your ownership of the custom domain. For more information, see Request a Public Certificate. When you create your domain configuration, use this public certificate as your validation certificate.

Creating a domain configuration

You create a configurable endpoint on a custom domain by using the CreateDomainConfiguration API. A domain configuration for a custom domain consists of the following:

- `domainConfigurationName`

  A user-defined name that identifies the domain configuration. Domain configuration names starting with `IoT:` are reserved for default endpoints and can't be used. Also, this value must be unique to your AWS Region.

- `domainName`

The FQDN that your devices use to connect to AWS IoT Core. AWS IoT Core leverages the server name indication (SNI) TLS extension to apply domain configurations. Devices must use this extension when connecting and pass a server name that is identical to the domain name that is specified in the domain configuration.

- `serverCertificateArns`

  The ARN of the server certificate chain that you registered with ACM. AWS IoT Core currently supports only one server certificate.

- `validationCertificateArn`

  The ARN of the public certificate that you generated in ACM to validate ownership of your custom domain. This argument isn't required if you use a publicly signed or ACM-generated server certificate.

- `defaultAuthorizerName (optional)`

  The name of the custom authorizer to use on the endpoint.

- `allowAuthorizerOverride`

  A Boolean value that specifies whether devices can override the default authorizer by specifying a different authorizer in the HTTP header of the request. This value is required if a value for `defaultAuthorizerName` is specified.

- `serviceType`

  AWS IoT Core currently supports only the `DATA` service type. When you specify `DATA`, AWS IoT returns an endpoint with an endpoint type of `iot:Data-ATS`.

- `TlsConfig` (optional)

  An object that specifies the TLS configuration for a domain. For more information, see ???.

- `serverCertificateConfig` (optional)

  An object that specifies the server certificate configuration for a domain. For more information, see ???.

The following AWS CLI command creates a domain configuration for **iot.example.com**.

```
aws iot create-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" --service-type "DATA"
```

```
--domain-name "iot.example.com" --server-certificate-arns serverCertARN --validation-certificate-arn validationCertArn
```

> **Note**
>
> After you create your domain configuration, it might take up to 60 minutes until AWS IoT Core serves your custom server certificates.

For more information, see ???.

Creating DNS records

After you register your server certificate chain and create your domain configuration, create a DNS record so that your custom domain points to an AWS IoT domain. This record must point to an AWS IoT endpoint of type `iot:Data-ATS`. You can get your endpoint by using the DescribeEndpoint API.

The following AWS CLI command shows how to get your endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

After you get your `iot:Data-ATS` endpoint, create a `CNAME` record from your custom domain to this AWS IoT endpoint. If you create multiple custom domains in the same AWS account, alias them to this same `iot:Data-ATS` endpoint.

Troubleshooting

If you have trouble connecting devices to a custom domain, make sure that AWS IoT Core has accepted and applied your server certificate. You can verify that AWS IoT Core has accepted your certificate by using either the AWS IoT Core console or the AWS CLI.

To use the AWS IoT Core console, navigate to the **Domain configurations** page and select the domain configuration name. In the **Server certificate details** section, check the status and status details. If the certificate is invalid, replace it in ACM with a certificate that meets the certificate requirements listed in the previous section. If the certificate has the same ARN, AWS IoT Core will be pick it up and apply it automatically.

To check the certificate status by using the AWS CLI, call the DescribeDomainConfiguration API and specify your domain configuration name.

> **Note**
>
> If your certificate is invalid, AWS IoT Core will continue to serve the last valid certificate.

You can check which certificate is being served on your endpoint by using the following openssl command.

```
openssl s_client -connect custom-domain-name:8883 -showcerts -servername custom-domain-name
```

Managing domain configurations

This topic covers key operations for you to manage your domain configuration resources. You can also manage the lifecycles of existing configurations by using the following APIs: ListDomainConfigurations, DescribeDomainConfiguration, UpdateDomainConfiguration, and DeleteDomainConfiguration.

In this topic:

- Viewing domain configurations
- Updating domain configurations
- Deleting domain configurations
- Rotating certificates in custom domains

Viewing domain configurations

To return a paginated list of all domain configurations in your AWS account, use the ListDomainConfigurations API . You can see the details of a particular domain configuration using the DescribeDomainConfiguration API. This API takes a single `domainConfigurationName` parameter and returns the details of the specified configuration.

Example

Updating domain configurations

To update the status or the custom authorizer of your domain configuration, use the UpdateDomainConfiguration API. You can set the status to `ENABLED` or `DISABLED`. If you disable the domain configuration, devices connected to that domain receive an authentication error.

Currently you can't update the server certificate in your domain configuration. To change the certificate of a domain configuration, you must delete and recreate it.

Example

Deleting domain configurations

Before you delete a domain configuration, use the UpdateDomainConfiguration API to set the status to `DISABLED`. This helps you avoid accidentally deleting the endpoint. After you disable the domain configuration, delete it by using the DeleteDomainConfiguration API. You must place AWS-managed domains in `DISABLED` status for 7 days before you can delete them. You can place custom domains in `DISABLED` status and then delete them at once.

Example

After you delete a domain configuration, AWS IoT Core no longer serves the server certificate associated with that custom domain.

Rotating certificates in custom domains

You may need to periodically replace your server certificate with an updated certificate. The rate at which you do this depends on the validity period of your certificate. If you generated your server certificate by using AWS Certificate Manager (ACM), you can set the certificate to renew automatically. When ACM renews your certificate, AWS IoT Core automatically picks up the new certificate. You don't have to perform any additional action. If you imported your server certificate from a different source, you can rotate it by reimporting it to ACM. For information about reimporting certificates, see Reimport a certificate.

> **Note**
>
> AWS IoT Core only picks up certificate updates under the following conditions.
>
> - The new certificate has the same ARN as the old one.
> - The new certificate has the same signing algorithm, common name, or subject alternative name as the old one.

Configuring TLS settings in domain configurations

AWS IoT Core provides predefined security polices for you to customize your Transport Layer Security (TLS) settings for TLS 1.2 and TLS 1.3 in domain configurations. A security policy is a combination of TLS protocols and their ciphers that determine the supported protocols and ciphers during TLS negotiations between a client and a server. With the supported security policies, you can manage your devices' TLS settings with more flexibility, apply the most up-to-date security measures when connecting new devices, and maintain consistent TLS configurations for existing devices.

The following table describes the security policies, their TLS versions, and supported regions:

Security policy name	Supported AWS Regions
IoTSecurityPolicy_ TLS13_1_3_2022_10	All AWS Regions
IoTSecurityPolicy_ TLS13_1_2_2022_10	All AWS Regions
IoTSecurityPolicy_ TLS12_1_2_2022_10	All AWS Regions
IoTSecurityPolicy_ TLS12_1_0_2016_01	ap-east-1, ap-northeast-2, ap-south-1, ap-southeast-2, ca-central-1, cn-north-1, cn-northwest-1, eu-north-1, eu-west-2, eu-west-3, me-south-1, sa-east-1, us-east-2, us-west-1
IoTSecurityPolicy_ TLS12_1_0_2015_01	ap-northeast-1, ap-southeast-1, eu-central-1, eu-west-1, us-east-1, us-west-2

The names of the security policies in AWS IoT Core include version information based on the year and month that they were released. If you create a new domain configuration, the security policy will default to `IoTSecurityPolicy_TLS13_1_2_2022_10`. For a complete table of security policies with details of protocols, TCP ports, and ciphers, see Security polices. AWS IoT Core doesn't support custom security policies. For more information, see ???.

To configure TLS settings in domain configurations, you can use the AWS IoT console or the AWS CLI.

Contents

Configure TLS settings in domain configurations (console)

To configure TLS settings using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. To configure TLS settings when you create a new domain configuration, follow these steps.

   1. In the left navigation pane, choose **Domain configurations**, and then choose **Create domain configuration**.
   2. In the **Create domain configuration** page, in the **Custom domain settings - *optional*** section, choose a security policy from **Select security policy**.
   3. Follow the widget and complete the rest of the steps. Choose **Create domain configuration**.
3. To update TLS settings in an existing domain configuration, follow these steps.

   1. In the left navigation pane, choose **Domain configurations**, and then choose a domain configuration.
   2. In the **Domain configuration details** page, choose **Edit**. Then, in the **Custom domain settings - *optional*** section, under **Select security policy**, choose a security policy.
   3. Choose **Update domain configuration**.

For more information, see Create a domain configuration and Manage domain configurations.

Configure TLS settings in domain configurations (CLI)

You can use the **create-domain-configuration** and **update-domain-configuration** CLI commands to configure your TLS settings in domain configurations.

1. To specify TLS settings using the **create-domain-configuration** CLI command:

   ```
   aws iot create-domain-configuration \
       --domain-configuration-name domainConfigurationName \
       --tls-config securityPolicy=IoTSecurityPolicy_TLS13_1_2_2022_10
   ```

   The output of this command can look like the following:

   ```
   {
   "domainConfigurationName": "test",
   "domainConfigurationArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
   test/34ga9"
   ```

```
}
```

If you create a new domain configuration without specifying the security policy, the value will default to: `IoTSecurityPolicy_TLS13_1_2_2022_10`.

2. To describe TLS settings using the [describe-domain-configuration](#) CLI command:

```
aws iot describe-domain-configuration \
    --domain-configuration-name domainConfigurationName
```

This command can return the domain configuration details that include the TLS settings like the following:


```
{
 "tlsConfig": {
 "securityPolicy": "IoTSecurityPolicy_TLS13_1_2_2022_10"
 },
 "domainConfigurationStatus": "ENABLED",
 "serviceType": "DATA",
 "domainType": "AWS_MANAGED",
 "domainName": "d1234567890abcdefghij-ats.iot.us-west-2.amazonaws.com",
 "serverCertificates": [],
 "lastStatusChangeDate": 1678750928.997,
 "domainConfigurationName": "test",
 "domainConfigurationArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
test/34ga9"
}
```


3. To update TLS settings using the [update-domain-configuration](#) CLI command:

```
aws iot update-domain-configuration \
    --domain-configuration-name domainConfigurationName \
    --tls-config securityPolicy=IoTSecurityPolicy_TLS13_1_2_2022_10
```

The output of this command can look like the following:


```
{
"domainConfigurationName": "test",
"domainConfigurationArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
test/34ga9"
}
```

4. To update the TLS settings for your ATS endpoint, run the **update-domain-configuration** CLI command. The domain configuration name for your ATS endpoint is `iot:Data-ATS`.

```
aws iot update-domain-configuration \
    --domain-configuration-name "iot:Data-ATS" \
    --tls-config securityPolicy=IoTSecurityPolicy_TLS13_1_2_2022_10
```

   The output of the command can look like the following:

```
{
"domainConfigurationName": "iot:Data-ATS",
"domainConfigurationArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
iot:Data-ATS"
}
```

For more information, see CreateDomainConfiguration and UpdateDomainConfiguration in the *AWS API Reference*.

Server certificate configuration for OCSP stapling

AWS IoT Core supports Online Certificate Status Protocol (OCSP) stapling for server certificate, also known as server certificate OCSP stapling, or OCSP stapling. It is a security mechanism used to check the revocation status on the server certificate in a Transport Layer Security (TLS) handshake. OCSP stapling in AWS IoT Core lets you add an additional layer of verification to your custom domain's server certificate validity.

You can enable server certificate OCSP stapling in AWS IoT Core to check the validity of the certificate by querying the OCSP responder periodically. The OCSP stapling setting is part of the process to create or update a domain configuration with a custom domain. OCSP stapling checks for revocation status on the server certificate continuously. This helps verify that any certificates that have been revoked by the CA are no longer trusted by the clients connecting to your custom domains. For more information, see ???.

Server certificate OCSP stapling provides real-time revocation status check, reduces the latency associated with checking the revocation status, and improves privacy and reliability of secure connections. For more information about the benefits of using OCSP stapling, see ???.

> **Note**
>
> This feature is not available in AWS GovCloud (US) Regions.

In this topic:

What is OCSP?

The Online Certificate Status Protocol (OCSP) aids in providing a server certificate's revocation status for a Transport Layer Security (TLS) handshake.

Key concepts

The following key concepts provide details about the Online Certificate Status Protocol (OCSP).

OCSP

OCSP is used to check the certificate revocation status during the Transport Layer Security (TLS) handshake. OCSP allows for real-time validation of certificates. This confirms that the certificate hasn't been revoked or expired since it was issued. OCSP is also more scalable compared with traditional Certificate Revocation Lists (CRLs). OCSP responses are smaller and can be efficiently generated, making them more suitable for large-scale Private Key Infrastructures (PKIs).

OCSP responder

An OCSP responder (also known as OCSP server) receives and responds to OCSP requests from clients that seek to verify the revocation status of certificates.

Client-side OCSP

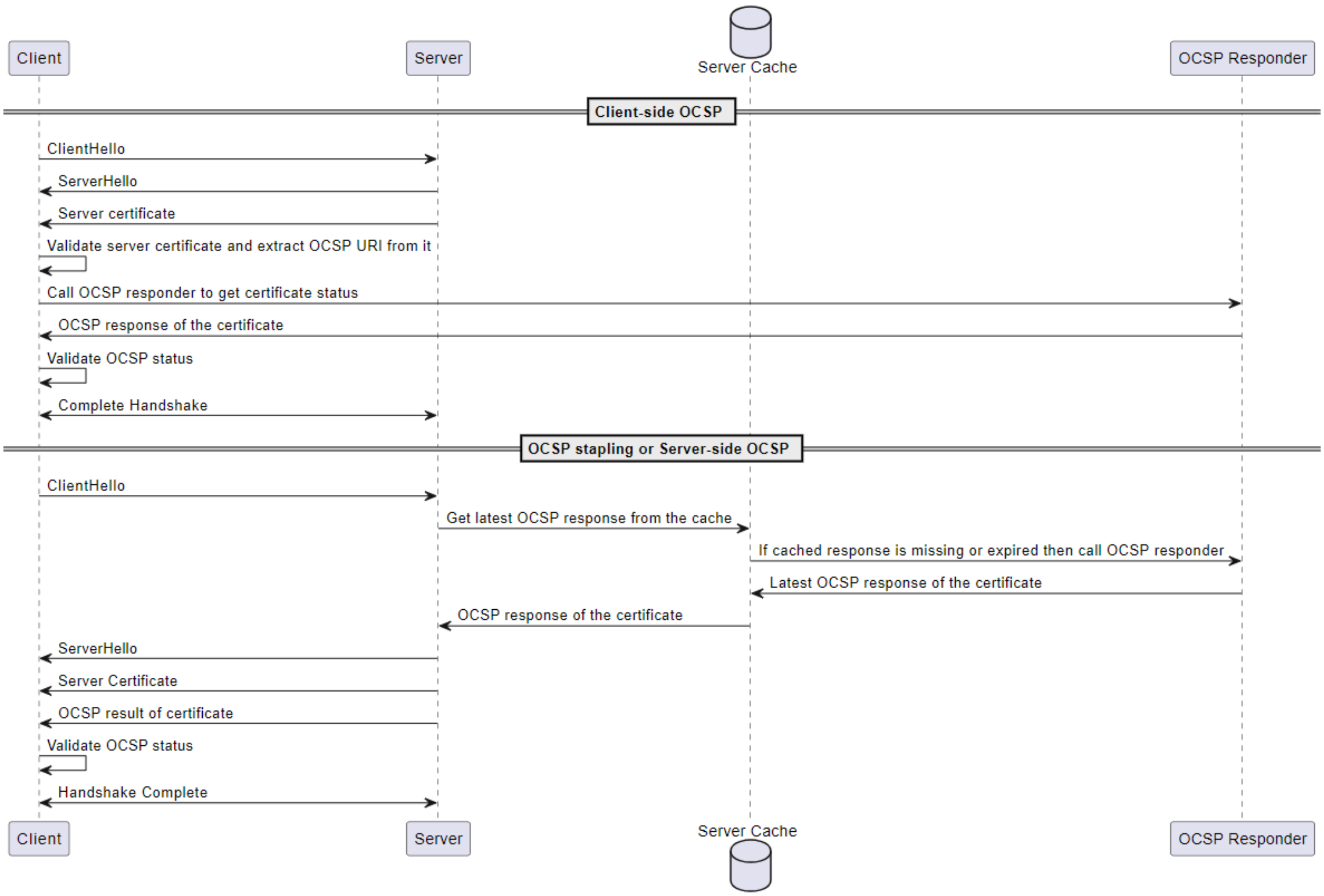
In client-side OCSP, the client uses OCSP to contact an OCSP responder to check the certificate's revocation status during the TLS handshake.

Server-side OCSP

In server-side OCSP (also known as OCSP stapling), the server is enabled (rather than the client) to make the request to the OCSP responder. The server staples the OCSP response to the certificate and returns it to the client during the TLS handshake.

OCSP diagrams

The following diagram illustrates how client-side OCSP and server-side OCSP work.



Client-side OCSP

1. The client sends a `ClientHello` message to initiate the TLS handshake with the server.
2. The server receives the message and responds with a `ServerHello` message. The server also sends the server certificate to the client.
3. The client validates the server certificate and extracts an OCSP URI from it.
4. The client sends a certificate revocation check request to the OCSP responder.

5. The OCSP responder sends an OCSP response.
6. The client validates the certificate status from the OCSP response.
7. The TLS handshake is completed.

Server-side OCSP

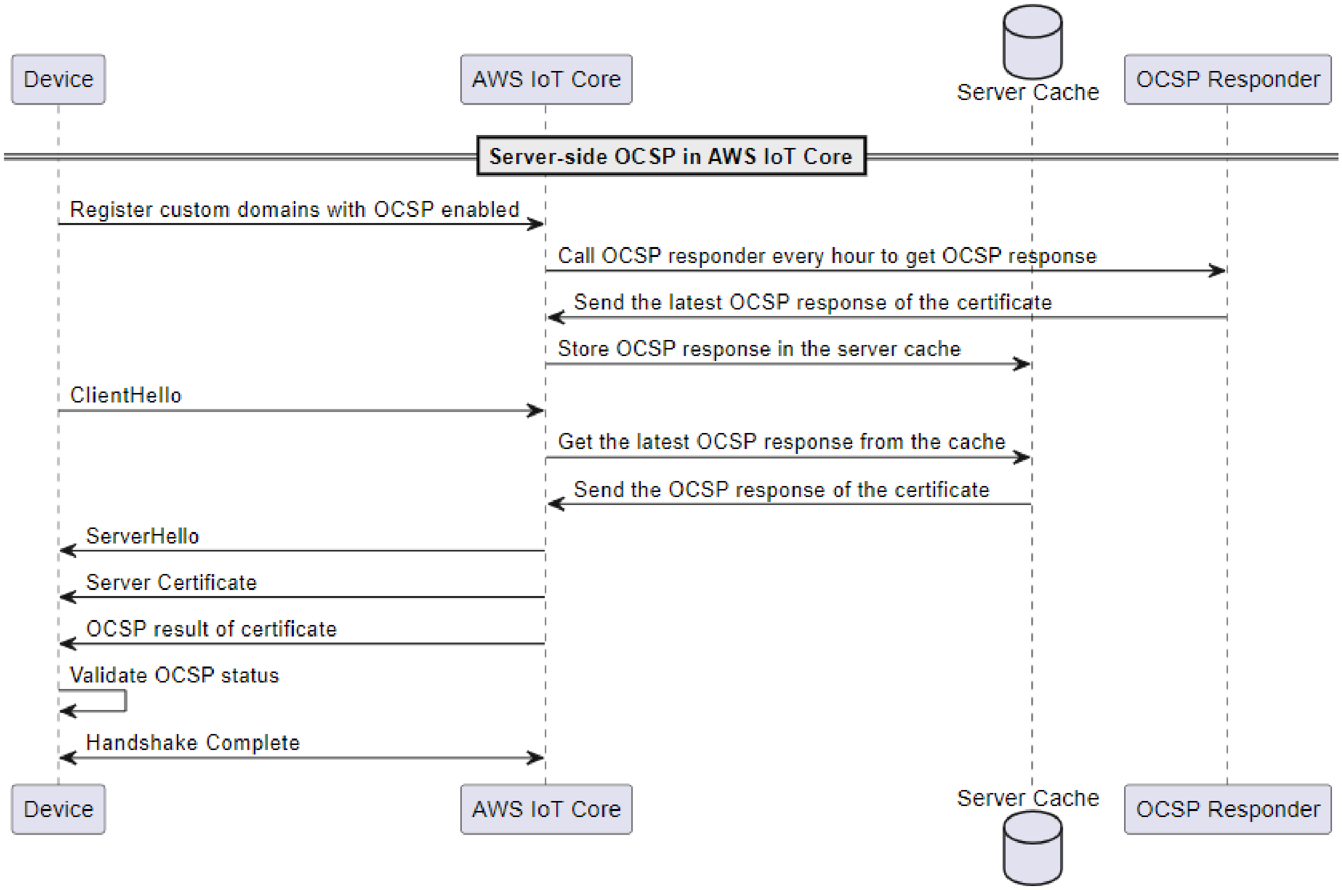
1. The client sends a `ClientHello` message to initiate the TLS handshake with the server.
2. The server receives the message and gets the latest cached OCSP response. If the cached response is missing or expired, the server will call the OCSP responder for certificate status.
3. The OCSP responder sends an OCSP response to the server.
4. The server sends a `ServerHello` message. The server also sends the server certificate and the certificate status to the client.
5. The client validates the OCSP certificate status.
6. The TLS handshake is completed.

How OCSP stapling works

OCSP stapling is used during the TLS handshake between the client and the server to check the server certificate revocation status. The server makes the OCSP request to the OCSP responder and staples the OCSP responses to the certificates returned to the client. By having the server make the request to the OCSP responder, the responses can be cached and then used multiple times for many clients.

How OCSP stapling works in AWS IoT Core

The following diagram shows how server-side OCSP stapling works in AWS IoT Core.



1. The device needs to be registered with custom domains with OCSP stapling enabled.
2. AWS IoT Core calls OCSP responder every hour to get the certificate status.
3. The OCSP responder receives the request, sends the latest OCSP response, and stores the cached OCSP response.
4. The device sends a `ClientHello` message to initiate the TLS handshake with AWS IoT Core.
5. AWS IoT Core gets the latest OCSP response from the server cache, which responds with an OCSP response of the certificate.
6. The server sends a `ServerHello` message to the device. The server also sends the server certificate and the certificate status to the client.
7. The device validates the OCSP certificate status.
8. The TLS handshake is completed.

Benefits of using OCSP stapling compared to client-side OCSP checks

A few advantages of using server certificate OCSP stapling include the following:

Improved privacy

Without OCSP stapling, the client's device can expose information to third-party OCSP responders, potentially compromising user privacy. OCSP stapling mitigates this issue by having the server obtain the OCSP response and deliver it directly to the client.

Improved reliability

OCSP stapling can improve the reliability of secure connections because it reduces the risk of OCSP server outages. When OCSP responses are stapled, the server includes the most recent response with the certificate. This is so that clients have access to the revocation status even if the OCSP responder is temporarily unavailable. OCSP stapling helps mitigate these problems because the server fetches OCSP responses periodically and includes the cached responses in the TLS handshake. This reduces reliance on the real-time availability of OCSP responders.

Reduced server load

OCSP stapling offloads the burden of responding to OCSP requests from OCSP responders to the server. This can help distribute the load more evenly, making the certificate validation process more efficient and scalable.

Reduced latency

OCSP stapling reduces the latency associated with checking the revocation status of a certificate during the TLS handshake. Instead of the client having to query an OCSP server separately, the server sends the request and attaches the OCSP response with the server certificate during the handshake.

Enabling server certificate OCSP in AWS IoT Core

To enable server certificate OCSP stapling in AWS IoT Core, create a domain configuration for a custom domain or update an existing custom domain configuration. For general information about creating a domain configuration with a custom domain, see ???.

Use the following instructions to enable OCSP server stapling using AWS Management Console or AWS CLI.

Console

To enable server certificate OCSP stapling using the AWS IoT console:

1. In the navigation menu, choose **Settings**, and then choose **Create domain configuration**, or choose an existing domain configuration for a custom domain.
2. If you choose to create a new domain configuration in the previous step, you will see the **Create domain configuration** page. In the **Domain configuration properties** section, choose **Custom domain**. Enter the information to create a domain configuration.

   If you choose to update an existing domain configuration for a custom domain, you will see the **Domain configuration details** page. Choose **Edit**.
3. To enable OCSP server stapling, choose **Enable server certificate OCSP stapling** in the **Server certificate configurations** subsection.
4. Choose **Create domain configuration** or **Update domain configuration**.

AWS CLI

To enable server certificate OCSP stapling using AWS CLI:

1. If you create a new domain configuration for a custom domain, the command to enable the OCSP server stapling can look like the following:

```
aws iot create-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" \
        --server-certificate-arns arn:aws:iot:us-
east-1:123456789012:cert/
f8c1e5480266caef0fdb1bf97dc1c82d7ba2d3e2642c5f25f5ba364fc6b79ba3 \
        --server-certificate-config "enableOCSPCheck=true|false"
```

2. If you update an existing domain configuration for a custom domain, the command to enable the OCSP server stapling can look like the following:

```
aws iot update-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" \
        --server-certificate-arns arn:aws:iot:us-
east-1:123456789012:cert/
f8c1e5480266caef0fdb1bf97dc1c82d7ba2d3e2642c5f25f5ba364fc6b79ba3 \
```

```
        --server-certificate-config "enableOCSPCheck=true|false"
```

For more information, see CreateDomainConfiguration and UpdateDomainConfiguration from the AWS IoT API Reference.

Configuring server certificate OCSP for private endpoints in AWS IoT Core

OCSP for private endpoints lets you use your private OCSP resources within your Amazon Virtual Private Cloud (Amazon VPC) for AWS IoT Core operations. The process involves setting up a Lambda function that acts as an OCSP responder. The Lambda function might use your private OCSP resources to craft OCSP responses that AWS IoT Core will use.

Lambda function

Before you configure server OCSP for a private endpoint, create a Lambda function that acts as a Request for Comments (RFC) 6960-compliant Online Certificate Status Protocol (OCSP) responder, supporting basic OCSP responses. The Lambda function accepts a base64-encoding of the OCSP request in the Distinguished Encoding Rules (DER) format. The Lambda function's response is also a base64-encoded OCSP response in the DER format. The response size must not exceed 4 kilobytes (KiB). The Lambda function must be in the same AWS account and AWS Region as the domain configuration. The following are example Lambda functions.

Example Lambda functions

JavaScript

```
import * as pkijs from 'pkijs';
console.log('Loading function');

export const handler = async (event, context) => {
    const requestBytes = decodeBase64(event);
    const ocspRequest = pkijs.OCSPRequest.fromBER(requestBytes);

    console.log("Here is a better look at the OCSP request");
    console.log(ocspRequest.toJSON());

    const ocspResponse = getOcspResponse();

    console.log("Here is a better look at the OCSP response");
    console.log(ocspResponse.toJSON());
```

```
    const responseBytes = ocspResponse.toSchema().toBER();
    return encodeBase64(responseBytes);
};

function getOcspResponse() {
    const responseString = "MIIC/
woBAKCCAvgwggL0BgkrBgEFBQcwAQEEggLlMIIC4TCByqFkMGIxJzAlBgNVBAoMHlJpY2hhcmQncyBEaXNjb3VudCBMY
p5w7WOtPjp3otNtVgIBAYAAGA8yMDI0MDQyMzE4NTMyNVowDQYJKoZIhvcNAQELBQADggIBAJFRyjDAHfazNejo704Ra
+s82R1spDarr3k7Pzkod9jJhwsZ2YgushlS4Npfe4lHCdwFyZR75WXrW55aXFddy03KLz01ZLNYyxkleW3f5dgrUcRU3
DEBiyS7ZsyhKo6igWU/SY7YMSKgwBvFsqSDcOa/hRYQkxWKWJ19gcz8CIkWN7NvfIxCs6VrAdzEJwmE7y3v
+jdfhxW9JmI4xStE4K0tAR9vVOOfKs7NvxXj7oc9pCSG60xl96kaEE6PaY1YsfNTsKQ7pyCJ0s7/2q
+ieZ4AtNyzw1XBadPzPJNv6E0LvI24yQZqN5wACvtut5prMMRxAHbOy
+abLZR58wloFSELtGJ7UD96LFv1GgtC5s
+2QlzPc4bEEof7Lo1EISt3j2ibNch8LxhqTQ4ufrbhsMkpSOTFYEJVMJF6aKj/OGXBUUqgc0Jx6jjJXNQd
+l5KCY9pQFeb/wVUYC6mYqZOkNNMMJxPbHHbFnqb68yO+g5BE9011N44YXoPVJYoXxBLFX+OpRu9cqPkT9/
vlkKd+SYXQknwZ81agKzhf1HsBKabtJwNVMlBKaI8g5UGa7Bxi6ewH3ezdWiERRUK7F56OM53wto/";
    const responseBytes = decodeBase64(responseString);
    return pkijs.OCSPResponse.fromBER(responseBytes);
}

function decodeBase64(input) {
    const binaryString = atob(input);

    const byteArray = new Uint8Array(binaryString.length);
    for (var i = 0; i < binaryString.length; i++) {
        byteArray[i] = binaryString.charCodeAt(i);
    }

    return byteArray.buffer;
}

function encodeBase64(buffer) {
    var binary = '';
    const bytes = new Uint8Array( buffer );
    const len = bytes.byteLength;

    for (var i = 0; i < len; i++) {
        binary += String.fromCharCode( bytes[ i ] );
    }

    return btoa(binary);
}
```

Java

```
package com.example.ocsp.responder;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import org.bouncycastle.cert.ocsp.OCSPReq;
import org.bouncycastle.cert.ocsp.OCSPResp;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Base64;

public class LambdaResponderApplication implements RequestHandler<String, String> {
    @Override
    public String handleRequest(final String input, final Context context) {
        LambdaLogger logger = context.getLogger();

        byte[] decodedInput = Base64.getDecoder().decode(input);

        OCSPReq req;
        try {
            req = new OCSPReq(decodedInput);
        } catch (IOException e) {
            logger.log("Got an IOException creating the OCSP request: " +
 e.getMessage());
            throw new RuntimeException(e);
        }

        try {
            OCSPResp response = businessLogic.getMyResponse();
            String toReturn =
 Base64.getEncoder().encodeToString(response.getEncoded());
            return toReturn;
        } catch (Exception e) {
            logger.log("Got an exception creating the response: " + e.getMessage());
            return "";
        }
    }
}
```

Authorizing AWS IoT to invoke your Lambda function

In the process of creating the domain configuration with a Lambda OCSP responder, you must grant AWS IoT permission to invoke the Lambda function after the function is created. To grant the permission, you can use the add-permission CLI command.

Grant permission to your Lambda function using the AWS CLI

1. After inserting your values, enter the following command. Note that the `statement-id` value must be unique. Replace *Id-1234* with the exact value you have, otherwise, you might get a `ResourceConflictException` error.

   ```
   aws lambda add-permission  \
   --function-name "ocsp-function" \
   --principal "iot.amazonaws.com" \
   --action "lambda:InvokeFunction" \
   --statement-id "Id-1234" \
   --source-arn arn:aws:iot:us-east-1:123456789012:domainconfiguration/<domain-config-name>/*
   --source-account 123456789012
   ```

   IoT domain configuration ARNs will follow the following pattern. The service-generated suffix will not be known prior to creation time, thus you must replace the suffix with a `*`. You can update the permission once the domain configuration has been created and the exact ARN is known.

   `arn:aws:iot:use-east-1:123456789012:domainconfiguration/domain-config-name/service-generated-suffix`

2. If the command succeeds, it returns a permission statement, such as this example. You can continue to the next section to configure OCSP stapling for private endpoints.

   ```
   {
       "Statement": "{\"Sid\":\"Id-1234\",\"Effect\":\"Allow\",\"Principal
   \":{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
   \",\"Resource\":\"arn:aws:lambda:us-east-1:123456789012:function:ocsp-
   function\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:iot:us-
   east-1:123456789012:domainconfiguration/domain-config-name/*\"}}}"
   }
   ```

If the command doesn't succeed, it returns an error, such as this example. You'll need to review and correct the error before you continue.

```
An error occurred (AccessDeniedException) when calling the AddPermission operation:
 User: arn:aws:iam::57EXAMPLE833:user/EXAMPLE-1 is not authorized to perform:
 lambda:AddPer
mission on resource: arn:aws:lambda:us-east-1:123456789012:function:ocsp-function
```

Configuring server OCSP stapling for private endpoints

Console

To configure server certificate OCSP stapling using the AWS IoT console:

1. From the navigation menu, choose **Settings**, and then choose **Create domain configuration**, or choose an existing domain configuration for a custom domain.
2. If you choose to create a new domain configuration in the previous step, you will see the **Create domain configuration** page. In the **Domain configuration properties** section, choose **Custom domain**. Enter the information to create a domain configuration.

   If you choose to update an existing domain configuration for a custom domain, you will see the **Domain configuration details** page. Choose **Edit**.
3. To enable OCSP server stapling, choose **Enable server certificate OCSP stapling** in the **Server certificate configurations** subsection.
4. Choose **Create domain configuration** or **Update domain configuration**.

AWS CLI

To configure server certificate OCSP stapling using AWS CLI:

1. If you create a new domain configuration for a custom domain, the command to configure server certificate OCSP for private endpoints can look like the following:

```
aws iot create-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" \
        --server-certificate-arns arn:aws:iot:us-
east-1:123456789012:cert/
f8c1e5480266caef0fdb1bf97dc1c82d7ba2d3e2642c5f25f5ba364fc6b79ba3 \
```

```
        --server-certificate-config "enableOCSPCheck=true,
 ocspAuthorizedResponderArn=arn:aws:acm:us-
east-1:123456789012:certificate/certificate_ID, ocspLambdaArn=arn:aws:lambda:us-
east-1:123456789012:function:my-function"
```

2. If you update an existing domain configuration for a custom domain, the command to configure server certificate OCSP for private endpoints can look like the following:

```
aws iot update-domain-configuration --domain-configuration-name
 "myDomainConfigurationName" \
        --server-certificate-arns arn:aws:iot:us-
east-1:123456789012:cert/
f8c1e5480266caef0fdb1bf97dc1c82d7ba2d3e2642c5f25f5ba364fc6b79ba3 \
        --server-certificate-config "enableOCSPCheck=true,
 ocspAuthorizedResponderArn=arn:aws:acm:us-
east-1:123456789012:certificate/certificate_ID, ocspLambdaArn=arn:aws:lambda:us-
east-1:123456789012:function:my-function"
```

enableOCSPCheck

This is a Boolean value that indicates whether server OCSP stapling check is enabled or not. To enable server certificate OCSP stapling, this value must be true.

ocspAuthorizedResponderArn

This is a string value of the Amazon Resource Name (ARN) for an X.509 certificate stored in AWS Certificate Manager (ACM). If provided, AWS IoT Core will use this certificate to validate the signature of the received OCSP response. If not provided, AWS IoT Core will use the issuing certificate to validate the responses. The certificate must be in the same AWS account and AWS Region as the domain configuration. For more information about how to register your authorized responder certificate, see Import certificates into AWS Certificate Manager.

ocspLambdaArn

This is a string value of the Amazon Resource Name (ARN) for a Lambda function that acts as a Request for Comments (RFC) 6960-compliant (OCSP) responder, supporting basic OCSP responses. The Lambda function accepts a base64-encoding of the OCSP request which is encoded using the DER format. The Lambda function's response is also a base64-encoded OCSP response in the DER format. The response size must not exceed 4 kilobytes (KiB). The Lambda function must be in the same AWS account and AWS Region as the domain configuration.

For more information, see CreateDomainConfiguration and UpdateDomainConfiguration from the AWS IoT API Reference.

Important notes for using server certificate OCSP stapling in AWS IoT Core

When you use server certificate OCSP in AWS IoT Core, keep the following in mind:

1. AWS IoT Core supports only those OCSP responders that are reachable over public IPv4 addresses.
2. The OCSP stapling feature in AWS IoT Core doesn't support authorized responder. All OCSP responses must be signed by the CA that signed the certificate, and the CA must be part of the certificate chain of the custom domain.
3. The OCSP stapling feature in AWS IoT Core doesn't support custom domains that are created using self-signed certificates.
4. AWS IoT Core calls an OCSP responder every hour and caches the response. If the call to the responder fails, AWS IoT Core will staple the most recent valid response.
5. If `nextUpdateTime` is no longer valid, AWS IoT Core will remove the response from the cache, and TLS handshake will not include the OCSP response data until the next successful call to the OCSP responder. This can happen when the cached response has expired before the server gets a valid response from the OCSP responder. The value of `nextUpdateTime` suggests that the OCSP response will be valid until this time. For more information about `nextUpdateTime`, see ???.
6. Sometimes, AWS IoT Core fails to receive the OCSP response or removes the existing OCSP response because it's expired. If situations like these happen, AWS IoT Core will continue to use the server certificate provided by the custom domain without the OCSP response.
7. The size of the OCSP response cannot exceed 4 KiB.

Troubleshooting server certificate OCSP stapling in AWS IoT Core

AWS IoT Core emits the `RetrieveOCSPStapleData.Success` metric and the `RetrieveOCSPStapleData` log entries to CloudWatch. The metric and the log entries can help detect issues related to retrieving OCSP responses. For more information, see ??? and ???.

Connect to AWS IoT FIPS endpoints

AWS IoT provides endpoints that support the Federal Information Processing Standard (FIPS) 140-2. FIPS compliant endpoints are different from standard AWS endpoints. To interact with

AWS IoT in a FIPS-compliant manner, you must use the endpoints described below with your FIPS compliant client. The AWS IoT console is not FIPS compliant.

The following sections describe how to access the FIPS compliant AWS IoT endpoints by using the REST API, an SDK, or the AWS CLI.

Topics

AWS IoT Core - control plane endpoints

The FIPS compliant **AWS IoT Core - control plane** endpoints that support the AWS IoT operations and their related CLI commands are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Core - control plane** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant endpoint when you access the AWS IoT operations, use the AWS SDK or the REST API with the endpoint that is appropriate for your AWS Region.

To use the FIPS compliant endpoint when you run **aws iot** CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command.

AWS IoT Core - data plane endpoints

The FIPS compliant **AWS IoT Core - data plane** endpoints are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Core - data plane** service, and look up the endpoint for your AWS Region.

You can use the FIPS compliant endpoint for your AWS Region with a FIPS compliant client by using the AWS IoT Device SDK and providing the endpoint to the SDK's connection function in place of your account's default **AWS IoT Core - data plane** endpoint. The connection function is specific to the AWS IoT Device SDK. For an example of a connection function, see the Connection function in the AWS IoT Device SDK for Python.

> **Note**
>
> AWS IoT doesn't support AWS account-specific **AWS IoT Core - data plane** endpoints that are FIPS-compliant. Service features that require an AWS account-specific endpoint in the Server Name Indication (SNI) can't be used. FIPS-compliant **AWS IoT Core - data plane** endpoints can't support Multi-Account Registration Certificates, Custom Domains, Custom Authorizers, and Configurable Endpoints (including supported TLS policies).

AWS IoT Core - credential provider endpoints

The FIPS compliant **AWS IoT Core - credential provider** endpoints are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Core - credential provider** service, and look up the endpoint for your AWS Region.

> **Note**
>
> AWS IoT doesn't support AWS account-specific **AWS IoT Core - credential provider** endpoints that are FIPS-compliant. Service features that require an AWS account-specific endpoint in the Server Name Indication (SNI) can't be used. FIPS-compliant **AWS IoT Core - credential provider** endpoints can't support Multi-Account Registration Certificates, Custom Domains, Custom Authorizers, and Configurable Endpoints (including supported TLS policies).

AWS IoT Device Management - jobs data endpoints

The FIPS compliant **AWS IoT Device Management - jobs data** endpoints are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Device Management - jobs data** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - jobs data** endpoint when you run **aws iot-jobs-data** CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

We recommend using `Data-ATS` instead of `iot:Jobs`. `iot:Data-ATS` supports dual-stack endpoints (IPv4 and IPv6) while `iot:Jobs` supports only IPv4.

You can use the FIPS compliant endpoint for your AWS Region with a FIPS compliant client by using the AWS IoT Device SDK and providing the endpoint to the SDK's connection function in place of your account's default **AWS IoT Device Management - jobs data** endpoint. The connection function is specific to the AWS IoT Device SDK. For an example of a connection function, see the Connection function in the AWS IoT Device SDK for Python.

AWS IoT Device Management - Fleet Hub endpoints

The FIPS compliant **AWS IoT Device Management - Fleet Hub** endpoints to use with Fleet Hub for AWS IoT Device Management CLI commands are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Device Management - Fleet Hub** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - Fleet Hub** endpoint when you run **aws iotfleethub** CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

AWS IoT Device Management - secure tunneling endpoints

The FIPS compliant **AWS IoT Device Management - secure tunneling** endpoints for the AWS IoT secure tunneling API and the corresponding CLI commands are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Device Management - secure tunneling** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant **AWS IoT Device Management - secure tunneling** endpoint when you run **aws iotsecuretunneling** CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command. You can also use the REST API with this endpoint.

AWS IoT Device Management - Managed Integrations endpoints

The FIPS compliant **control plane** endpoints that support the managed integrations operations and their related AWS CLI commands are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Device Management - Managed integrations** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant endpoint when you access the managed integrations operations, use the AWS SDK or the REST API with the endpoint that is appropriate for your AWS Region.

To use the FIPS compliant endpoint when you run managed integrations CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command.

Managing devices with AWS IoT

AWS IoT provides a registry that helps you manage *things*. A thing is a representation of a specific device or logical entity. It can be a physical device or sensor (for example, a light bulb or a switch on a wall). It can also be a logical entity like an instance of an application or physical entity that does not connect to AWS IoT but is related to other devices that do (for example, a car that has engine sensors or a control panel).

Information about a thing is stored in the registry as JSON data. Here is an example thing:

```
{
     "version": 3,
    "thingName": "MyLightBulb",
    "defaultClientId": "MyLightBulb",
    "thingTypeName": "LightBulb",
    "attributes": {
        "model": "123",
        "wattage": "75"
    }
}
```

Things are identified by a name. Things can also have attributes, which are name-value pairs you can use to store information about the thing, such as its serial number or manufacturer.

A typical device use case involves the use of the thing name as the default MQTT client ID. Although we don't enforce a mapping between a thing's registry name and its use of MQTT client IDs, certificates, or shadow state, we recommend you choose a thing name and use it as the MQTT client ID for both the registry and the Device Shadow service. This provides organization and convenience to your IoT fleet without removing the flexibility of the underlying device certificate model or shadows.

You don't need to create a thing in the registry to connect a device to AWS IoT. Adding things to the registry allows you to manage and search for devices more easily.

Thing registry data (including attributes, thing types, and group memberships) can also be dynamically retrieved in the rules engine to use in message processing and routing. For more information, see `get_registry_data`.

Managing things with the registry

You use the AWS IoT console, AWS IoT API, or the AWS CLI to interact with the registry. The following sections show how to use the CLI to work with the registry.

When naming your thing objects:

- Don't use personally identifiable information in your thing name. The thing name can appear in unencrypted communications and reports.

Topics

Create a thing

The following command shows how to use the AWS IoT **CreateThing** command from the CLI to create a thing. You can't change a thing's name after you create it. To change a thing's name, create a new thing, give it the new name, and then delete the old thing.

```
$ aws iot create-thing \
    --thing-type-name "MyLightBulb" \
    --attribute-payload "{\"attributes\": {\"wattage\":\"75\", \"model\":\"123\"}}"
```

The **CreateThing** command displays the name and Amazon Resource Name (ARN) of your new thing:

```
{
    "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/MyLightBulb",
    "thingName": "MyLightBulb",
    "thingId": "12345678abcdefgh12345678ijklmnop12345678"
}
```


> **Note**
>
> We don't recommend using personally identifiable information in your thing names.

For more information, see create-thing from the AWS CLI Command Reference.

List things

You can use the **ListThings** command to list all things in your account:

```
$ aws iot list-things
```


```
{
    "things": [
       {
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MyLightBulb"
        },
        {
            "attributes": {
                "numOfStates":"3"
             },
            "version": 11,
            "thingName": "MyWallSwitch"
        }
    ]
}
```


You can use the **ListThings** command to search for all things of a specific thing type:

```
$  aws iot list-things --thing-type-name "LightBulb"
```

```
{
    "things": [
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MyRGBLight"
        },
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MySecondLightBulb"
        }
    ]
}
```

You can use the **ListThings** command to search for all things that have an attribute with a specific value. This command searches up to three attributes.

```
$  aws iot list-things --attribute-name "wattage" --attribute-value "75"
```

```

{
    "things": [
        {
            "thingTypeName": "StopLight",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 3,
            "thingName": "MyLightBulb"
```

```
        },
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MyRGBLight"
        },
        {
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1,
            "thingName": "MySecondLightBulb"
        }
    ]
}
```

For more information, see list-things from the AWS CLI Command Reference.

Describe things

You can use the **DescribeThing** command to display more detailed information about a thing:

```
$ aws iot describe-thing --thing-name "MyLightBulb"
{
    "version": 3,
    "thingName": "MyLightBulb",
    "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/MyLightBulb",
    "thingId": "12345678abcdefgh12345678ijklmnop12345678",
    "defaultClientId": "MyLightBulb",
    "thingTypeName": "StopLight",
    "attributes": {
        "model": "123",
        "wattage": "75"
    }
}
```

You can also access this API within the rules engine using the inline function `get_registry_data()`. You can use this function to dynamically access and utilize thing registry information (including attributes, thing types, and group memberships) by calling `DescribeThing` and `ListThingGroupsForThing` APIs directly within AWS IoT rules, enabling real-time message processing and routing based on your device registry data. For more information, see get_registry_data.

For more information, see describe-thing from the AWS CLI Command Reference.

Update a thing

You can use the **UpdateThing** command to update a thing. This command updates only the thing's attributes. You can't change a thing's name. To change a thing's name, create a new thing, give it the new name, and then delete the old thing.

```
$ aws iot update-thing --thing-name "MyLightBulb" --attribute-payload "{\"attributes\":
 {\"wattage\":\"150\", \"model\":\"456\"}}"
```

The **UpdateThing** command does not produce output. You can use the **DescribeThing** command to see the result:

```
$ aws iot describe-thing --thing-name "MyLightBulb"
{
    "attributes": {
        "model": "456",
        "wattage": "150"
    },
    "version": 2,
    "thingName": "MyLightBulb"
}
```

For more information, see update-thing from the AWS CLI Command Reference.

Delete a thing

You can use the **DeleteThing** command to delete a thing:

```
$ aws iot delete-thing --thing-name "MyThing"
```

This command returns successfully with no error if the deletion is successful or you specify a thing that doesn't exist.

For more information, see delete-thing from the AWS CLI Command Reference.

Attach a principal to a thing

A physical device can use a principal to communicate with AWS IoT. A principal can be an X.509 certificate or an Amazon Cognito ID. You can associate a certificate or an Amazon Cognito ID with the thing in the registry that represents your device, by running the attach-thing-principal command.

To attach a certificate or an Amazon Cognito ID to your thing, use the attach-thing-principal command:

```
$ aws iot attach-thing-principal \
    --thing-name "MyLightBulb1" \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

To attach a certificate to your thing with an attachment type (exclusive attachment or non-exclusive attachment), use the **attach-thing-principal** command and specify a type in the `--thing-principal-type` field. An exclusive attachment means your IoT thing is the only thing attached to the certificate, and this certificate cannot be associated with any other things. An non-exclusive attachment means your IoT thing is attached to the certificate, and this certificate can be associated with other things. For more information, see ???.

> **Note**
>
> For the ??? feature, you can only use X.509 certificate as a principal.

```
$ aws iot attach-thing-principal \
    --thing-name "MyLightBulb2" \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847" \
    --thing-principal-type "EXCLUSIVE_THING"
```

If the attachment is successful, the **AttachThingPrincipal** command does not produce any output. To describe the attachment, use list-thing-principals-v2 CLI command.

For more information, see AttachThingPrincipal from the *AWS IoT Core API Reference*.

List things associated with a principal

To list the things associated with the specified principal, run the `list-principal-things` command. Note that this command doesn't list the attachment type between the thing and the certificate. To list the attachment type, use the `list-principal-things-v2` command. For more information, see ???.

```
$ aws iot list-principal-things \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
```

The output can look like the following.

```
{
    "things": [
        "MyLightBulb1",
        "MyLightBulb2"
    ]
}
```

For more information, see ListPrincipalThings from the *AWS IoT Core API Reference*.

List principals associated with a thing

To list the principals associated with the specified thing, run the `list-thing-principals` command. Note that this command doesn't list the attachment type between the thing and the certificate. To list the attachment type, use the `list-thing-principals-v2` command. For more information, see ???.

```
$ aws iot list-thing-principals \
    --thing-name "MyLightBulb1"
```

The output can look like the following.

```
{
    "principals": [
         "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8",
         "arn:aws:iot:us-
east-1:123456789012:cert/
1a234b39b4b68278f2e9d84bf97eac2cbf4a1c28b23ea29a44559b9bcf8d395b"
    ]
}
```

For more information, see ListThingPrincipals from the *AWS IoT Core API Reference*.

List things associated with a principal V2

To list the things associated with the specified certificate, along with the attachment type, run the `list-principal-things-v2` command. The attachment type refers to how the certificate is attached to the thing.

```
$ aws iot list-principal-things-v2 \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
```

The output can look like the following.

```
{
    "PrincipalThingObjects": [
        {
            "thingPrincipalType": "NON_EXCLUSIVE_THING",
            "thing": "arn:aws:iot:us-east-1:123456789012:thing/thing_1"
        },
        {
            "thingPrincipalType": "NON_EXCLUSIVE_THING",
            "thing": "arn:aws:iot:us-east-1:123456789012:thing/thing_2"
        }

    ]
}
```

For more information, see ListPrincipalThingsV2 from the *AWS IoT Core API Reference*.

List principals associated with a thing V2

To list the certificates associated with the specified thing, along with the attachment type, run the `list-thing-principals-V2` command. The attachment type refers to how the certificate is attached to the thing.

```
$ aws iot list-thing-principals-v2 \
    --thing-name "thing_1"
```

The output can look like the following.

```
{
    "ThingPrincipalObjects": [
        {
            "thingPrincipalType": "NON_EXCLUSIVE_THING",
            "principal": "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
        },
        {
            "thingPrincipalType": "NON_EXCLUSIVE_THING",
            "principal": "arn:aws:iot:us-
east-1:123456789012:cert/
1a234b39b4b68278f2e9d84bf97eac2cbf4a1c28b23ea29a44559b9bcf8d395b"
        }
    ]
}
```

For more information, see ListThingsPrincipalV2 from the *AWS IoT Core API Reference*.

Detach a principal from a thing

You can use the `DetachThingPrincipal` command to detach a certificate from a thing:

```
$ aws iot detach-thing-principal \
    --thing-name "MyLightBulb" \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
```

The **DetachThingPrincipal** command doesn't produce any output.

For more information, see detach-thing-principal from the *AWS IoT Core API Reference*.

Thing types

Thing types allow you to store description and configuration information that is common to all things associated with the same thing type. This simplifies the management of things in the registry. For example, you can define a LightBulb thing type. All things associated with the LightBulb thing type share a set of attributes: serial number, manufacturer, and wattage. When you create a thing of type LightBulb (or change the type of an existing thing to LightBulb) you can specify values for each of the attributes defined in the LightBulb thing type.

Although thing types are optional, their use makes it easier to discover things.

- Things with a thing type can have up to 50 attributes.
- Things without a thing type can have up to three attributes.
- A thing can be associated with only one thing type.
- There is no limit on the number of thing types you can create in your account.

You can't change a thing type name after it has been created. You can deprecate a thing type at any time to prevent new things from being associated with it. You can also delete thing types that have no things associated with them.

Topics:

Create a thing type

You can use the **CreateThingType** command to create a thing type:

```
$ aws iot create-thing-type

                --thing-type-name "LightBulb" --thing-type-properties
 "thingTypeDescription=light bulb type, searchableAttributes=wattage,model"
```

The **CreateThingType** command returns a response that contains the thing type and its ARN:

```
{
    "thingTypeName": "LightBulb",
    "thingTypeId": "df9c2d8c-894d-46a9-8192-9068d01b2886",
    "thingTypeArn": "arn:aws:iot:us-west-2:123456789012:thingtype/LightBulb"
}
```

List thing types

You can use the **ListThingTypes** command to list thing types:

```
$ aws iot list-thing-types
```

The **ListThingTypes** command returns a list of the thing types defined in your AWS account:

```
{
    "thingTypes": [
        {
            "thingTypeName": "LightBulb",
            "thingTypeProperties": {
                "searchableAttributes": [
                    "wattage",
                    "model"
                ],
                "thingTypeDescription": "light bulb type"
            },
            "thingTypeMetadata": {
                "deprecated": false,
                "creationDate": 1468423800950
            }
        }
    ]
}
```

Describe a thing type

You can use the **DescribeThingType** command to get information about a thing type:

```
$ aws iot describe-thing-type --thing-type-name "LightBulb"
```

The **DescribeThingType** command returns information about the specified type:

```
{
    "thingTypeProperties": {
        "searchableAttributes": [
            "model",
            "wattage"
        ],
        "thingTypeDescription": "light bulb type"
    },
    "thingTypeId": "df9c2d8c-894d-46a9-8192-9068d01b2886",
    "thingTypeArn": "arn:aws:iot:us-west-2:123456789012:thingtype/LightBulb",
    "thingTypeName": "LightBulb",
    "thingTypeMetadata": {
        "deprecated": false,
        "creationDate": 1544466338.399
    }
}
```

Associate a thing type with a thing

You can use the **CreateThing** command to specify a thing type when you create a thing:

```
$ aws iot create-thing --thing-name "MyLightBulb" --thing-type-name "LightBulb" --
attribute-payload "{\"attributes\": {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can use the **UpdateThing** command at any time to change the thing type associated with a thing:

```
$ aws iot update-thing --thing-name "MyLightBulb"
                --thing-type-name "LightBulb" --attribute-payload  "{\"attributes\":
 {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can also use the **UpdateThing** command to disassociate a thing from a thing type.

Update a thing type

You can use the **UpdateThingType** command to update a thing type when you create a thing:

```
$ aws iot create-thing --thing-name "MyLightBulb" --thing-type-name "LightBulb" --
attribute-payload "{\"attributes\": {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can use the **UpdateThing** command at any time to change the thing type associated with a thing:

```
$ aws iot update-thing --thing-name "MyLightBulb"
                --thing-type-name "LightBulb" --attribute-payload  "{\"attributes\":
 {\"wattage\":\"75\", \"model\":\"123\"}}"
```

You can also use the **UpdateThing** command to disassociate a thing from a thing type.

Deprecate a thing type

Thing types are immutable. They can't be changed after they are defined. You can, however, deprecate a thing type to prevent users from associating any new things with it. All existing things associated with the thing type are unchanged.

To deprecate a thing type, use the **DeprecateThingType** command:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType"
```

You can use the **DescribeThingType** command to see the result:

```
$ aws iot describe-thing-type --thing-type-name "StopLight":
```

```
{
    "thingTypeName": "StopLight",
    "thingTypeProperties": {
        "searchableAttributes": [
            "wattage",
            "numOfLights",
            "model"
        ],
```

```
        "thingTypeDescription": "traffic light type",
    },
    "thingTypeMetadata": {
        "deprecated": true,
        "creationDate": 1468425854308,
        "deprecationDate": 1468446026349
    }
}
```

Deprecating a thing type is a reversible operation. You can undo a deprecation by using the `--undo-deprecate` flag with the **DeprecateThingType** CLI command:

```
$ aws iot deprecate-thing-type --thing-type-name "myThingType" --undo-deprecate
```

You can use the **DescribeThingType** CLI command to see the result:

```
$ aws iot describe-thing-type --thing-type-name "StopLight":
```

```
{
    "thingTypeName": "StopLight",
    "thingTypeArn": "arn:aws:iot:us-east-1:123456789012:thingtype/StopLight",
    "thingTypeId": "12345678abcdefgh12345678ijklmnop12345678"
    "thingTypeProperties": {
        "searchableAttributes": [
            "wattage",
            "numOfLights",
            "model"
        ],
        "thingTypeDescription": "traffic light type"
    },
    "thingTypeMetadata": {
        "deprecated": false,
        "creationDate": 1468425854308,
    }
}
```

Delete a thing type

You can delete thing types only after they have been deprecated. To delete a thing type, use the **DeleteThingType** command:

```
$ aws iot delete-thing-type --thing-type-name "StopLight"
```

> **Note**
>
> Before you can delete a thing type, wait for five minutes after you deprecate it.

Static thing groups

Static thing groups allow you to manage several things at once by categorizing them into groups. Static thing groups contain a group of things that are managed by using the console, CLI, or the API. Dynamic thing groups, on the other hand, contain things that match a specified query. Static thing groups can also contain other static thing groups — you can build a hierarchy of groups. You can attach a policy to a parent group and it is inherited by its child groups, and by all of the things in the group and in its child groups. This makes control of permissions easy for large numbers of things.

> **Note**
>
> Thing group policies don't allow access to AWS IoT Greengrass data plane operations. To allow a thing access to an AWS IoT Greengrass data plane operation, add the permission to an AWS IoT policy that you attach to the thing's certificate. For more information, see Device authentication and authorization in the *AWS IoT Greengrass developer guide*.

Here are the things you can do with static thing groups:

- Create, describe or delete a group.
- Add a thing to a group, or to more than one group.
- Remove a thing from a group.
- List the groups you have created.
- List all child groups of a group (its direct and indirect descendants.)
- List the things in a group, including all the things in its child groups.
- List all ancestor groups of a group (its direct and indirect parents.)
- Add, delete or update the attributes of a group. (Attributes are name-value pairs you can use to store information about a group.)

- Attach or detach a policy to or from a group.
- List the policies attached to a group.
- List the policies inherited by a thing (by virtue of the policies attached to its group, or one of its parent groups.)
- Configure logging options for things in a group. See Configure AWS IoT logging.
- Create jobs that are sent to and executed on every thing in a group and its child groups. See AWS IoT Jobs.

> **Note**
>
> When a thing is attached to a static thing group to which an AWS IoT Core policy is attached to, the thing name must match the client ID.

Here are some limitations of static thing groups:

- A group can have at most one direct parent.
- If a group is a child of another group, specify this at the time it is created.
- You can't change a group's parent later, so be sure to plan your group hierarchy and create a parent group before you create any child groups it contains.
- The number of groups to which a thing can belong is limited.
- You can't add a thing to more than one group in the same hierarchy. (In other words, you can't add a thing to two groups that share a common parent.)
- You can't rename a group.
- Thing group names can't contain international characters, such as û, é and ñ.
- Don't use personally identifiable information in your thing group name. The thing group name can appear in unencrypted communications and reports.

Attaching and detaching policies to groups can enhance the security of your AWS IoT operations in a number of significant ways. The per-device method of attaching a policy to a certificate, which is then attached to a thing, is time consuming and makes it difficult to quickly update or change policies across a fleet of devices. Having a policy attached to the thing's group saves steps when it is time to rotate the certificates on a thing. And policies are dynamically applied to things when they change group membership, so you aren't required to re-create a complex set of permissions each time a device changes membership in a group.

Create a static thing group

Use the **CreateThingGroup** command to create a static thing group:

```
$ aws iot create-thing-group --thing-group-name LightBulbs
```

The **CreateThingGroup** command returns a response that contains the static thing group's name, ID, and ARN:

```
{
    "thingGroupName": "LightBulbs",
    "thingGroupId": "abcdefgh12345678ijklmnop12345678qrstuvwx",
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"
}
```

> **Note**
>
> We don't recommend using personally identifiable information in your thing group names.

Here is an example that specifies a parent of the static thing group when it is created:

```
$ aws iot create-thing-group --thing-group-name RedLights --parent-group-name
 LightBulbs
```

As before, the **CreateThingGroup** command returns a response that contains the static thing group's name,, ID, and ARN:

```
{
    "thingGroupName": "RedLights",
    "thingGroupId": "abcdefgh12345678ijklmnop12345678qrstuvwx",
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights",
}
```

> **Important**
>
> Keep in mind the following limits when creating thing group hierarchies:

- A thing group can have only one direct parent.
- The number of direct child groups a thing group can have is limited.
- The maximum depth of a group hierarchy is limited.
- The number of attributes a thing group can have is limited. (Attributes are name-value pairs you can use to store information about a group.) The lengths of each attribute name and each value are also limited.

Describe a thing group

You can use the **DescribeThingGroup** command to get information about a thing group:

```
$ aws iot describe-thing-group --thing-group-name RedLights
```

The **DescribeThingGroup** command returns information about the specified group:

```
{
    "thingGroupName": "RedLights",
    "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights",
    "thingGroupId": "12345678abcdefgh12345678ijklmnop12345678",
    "version": 1,
    "thingGroupMetadata": {
        "creationDate": 1478299948.882
        "parentGroupName": "Lights",
        "rootToParentThingGroups": [
            {
                "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
ShinyObjects",
                "groupName": "ShinyObjects"
            },
            {
                "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs",
                "groupName": "LightBulbs"
            }
        ]
    },
    "thingGroupProperties": {
        "attributePayload": {
            "attributes": {
```

```
                "brightness": "3400_lumens"
            },
        },
        "thingGroupDescription": "string"
    },
}
```

Add a thing to a static thing group

You can use the **AddThingToThingGroup** command to add a thing to a static thing group:

```
$ aws iot add-thing-to-thing-group --thing-name MyLightBulb --thing-group-name
 RedLights
```

The **AddThingToThingGroup** command does not produce any output.

> **Important**
>
> You can add a thing to a maximum of 10 groups. But you can't add a thing to more than one group in the same hierarchy. (In other words, you can't add a thing to two groups which share a common parent.)
> If a thing belongs to as many thing groups as possible, and one or more of those groups is a dynamic thing group, you can use the **overrideDynamicGroups** flag to make static groups take priority over dynamic groups.

Remove a thing from a static thing group

You can use the **RemoveThingFromThingGroup** command to remove a thing from a group:

```
$ aws iot remove-thing-from-thing-group --thing-name MyLightBulb --thing-group-name
 RedLights
```

The **RemoveThingFromThingGroup** command does not produce any output.

List things in a thing group

You can use the **ListThingsInThingGroup** command to list the things that belong to a group:

```
$ aws iot list-things-in-thing-group --thing-group-name LightBulbs
```

The **ListThingsInThingGroup** command returns a list of the things in the given group:

```
{
    "things":[
        "TestThingA"
    ]
}
```

With the `--recursive` parameter, you can list things belonging to a group and those in any of its child groups:

```
$ aws iot list-things-in-thing-group --thing-group-name LightBulbs --recursive
```

```
{
    "things":[
        "TestThingA",
        "MyLightBulb"
    ]
}
```

> **Note**
>
> This operation is eventually consistent. In other words, changes to the thing group might not be reflected at once.

List thing groups

You can use the **ListThingGroups** command to list your account's thing groups:

```
$ aws iot list-thing-groups
```

The **ListThingGroups** command returns a list of the thing groups in your AWS account:

```
{
```

```
    "thingGroups": [
        {
            "groupName": "LightBulbs",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"
        },
        {
            "groupName": "RedLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"
        },
        {
            "groupName": "RedLEDLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLEDLights"
        },
        {
            "groupName": "RedIncandescentLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
RedIncandescentLights"
        }
        {
            "groupName": "ReplaceableObjects",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
ReplaceableObjects"
        }
    ]
}
```

Use the optional filters to list those groups that have a given group as parent (`--parent-group`) or groups whose name begins with a given prefix (`--name-prefix-filter`.) The `--recursive` parameter allows you to list all children groups, not just direct child groups of a thing group:

```
$ aws iot list-thing-groups --parent-group LightBulbs
```

In this case, the **ListThingGroups** command returns a list of the direct child groups of the thing group defined in your AWS account:

```
{
    "childGroups":[
        {
            "groupName": "RedLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"
        }
    ]
```

```
}
```

Use the `--recursive` parameter with the **ListThingGroups** command to list all child groups of a thing group, not just direct children:

```
$ aws iot list-thing-groups --parent-group LightBulbs --recursive
```

The **ListThingGroups** command returns a list of all child groups of the thing group:

```
{
    "childGroups":[
        {
            "groupName": "RedLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"
        },
        {
            "groupName": "RedLEDLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLEDLights"
        },
        {
            "groupName": "RedIncandescentLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
RedIncandescentLights"
        }
    ]
}
```

Note

This operation is eventually consistent. In other words, changes to the thing group might not be reflected at once.

List groups for a thing

You can use the **ListThingGroupsForThing** command to list the direct groups that a thing belongs to:

```
$ aws iot list-thing-groups-for-thing --thing-name MyLightBulb
```

The **ListThingGroupsForThing** command returns a list of the direct thing groups that this thing belongs to:

```
{
    "thingGroups":[
        {
            "groupName": "LightBulbs",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs"
        },
        {
            "groupName": "RedLights",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/RedLights"
        },
        {
            "groupName": "ReplaceableObjects",
            "groupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
ReplaceableObjects"
        }
    ]
}
```

You can also access this API within the rules engine using the inline function `get_registry_data()`. You can use this function to dynamically access and utilize thing registry information (including attributes, thing types, and group memberships) by calling `DescribeThing` and `ListThingGroupsForThing` APIs directly within AWS IoT rules, enabling real-time message processing and routing based on your device registry data. For more information, see `get_registry_data`.

Update a static thing group

You can use the **UpdateThingGroup** command to update the attributes of a static thing group:

```
$ aws iot update-thing-group --thing-group-name "LightBulbs" --thing-group-properties
 "thingGroupDescription=\"this is a test group\", attributePayload=\"{\"attributes
\"={\"Owner\"=\"150\",\"modelNames\"=\"456\"}}"
```

The **UpdateThingGroup** command returns a response that contains the group's version number after the update:

```
{
    "version": 4
```

```
}
```

Note

The number of attributes that a thing can have is limited.

Delete a thing group

To delete a thing group, use the **DeleteThingGroup** command:

```
$ aws iot delete-thing-group --thing-group-name "RedLights"
```

The **DeleteThingGroup** command does not produce any output.

Important

If you try to delete a thing group that has child thing groups, you receive an error:

```
A client error (InvalidRequestException) occurred when calling the
 DeleteThingGroup
operation: Cannot delete thing group : RedLights when there are still child
 groups attached to it.
```

Before you delete the group, delete any child groups first.

You can delete a group that has child things, but any permissions granted to the things by membership in the group no longer apply. Before deleting a group that has a policy attached, check carefully that removing those permissions would not stop the things in the group from being able to function properly. Also, commands that show which groups a thing belongs to (for example, **ListGroupsForThing**) might continue to show the group while records in the cloud are being updated.

Attach a policy to a static thing group

You can use the **AttachPolicy** command to attach a policy to a static thing group and so, by extension, to all things in that group and things in any of its child groups:

```
$ aws iot attach-policy \
  --target "arn:aws:iot:us-west-2:123456789012:thinggroup/LightBulbs" \
  --policy-name "myLightBulbPolicy"
```

The **AttachPolicy** command does not produce any output

> **Important**
>
> You can attach a maximum number of two policies to a group.

> **Note**
>
> We don't recommend using personally identifiable information in your policy names.

The `--target` parameter can be a thing group ARN (as above), a certificate ARN, or an Amazon Cognito Identity. For more information about policies, certificates and authentication, see Authentication.

For more information, see AWS IoT Core policies.

Detach a policy from a static thing group

You can use the **DetachPolicy** command to detach a policy from a group and so, by extension, to all things in that group and things in any of its child groups:

```
$ aws iot detach-policy --target "arn:aws:iot:us-west-2:123456789012:thinggroup/
LightBulbs" --policy-name "myLightBulbPolicy"
```

The **DetachPolicy** command does not produce any output.

List the policies attached to a static thing group

You can use the **ListAttachedPolicies** command to list the policies attached to a static thing group:

```
$ aws iot list-attached-policies --target "arn:aws:iot:us-
west-2:123456789012:thinggroup/RedLights"
```

The `--target` parameter can be a thing group ARN (as above), a certificate ARN, or an Amazon Cognito identity.

Add the optional `--recursive` parameter to include all policies attached to the group's parent groups.

The **ListAttachedPolicies** command returns a list of policies:

```
{
    "policies": [
        "MyLightBulbPolicy"
        ...
    ]
}
```

List the groups for a policy

You can use the **ListTargetsForPolicy** command to list the targets, including any groups, that a policy is attached to:

```
$ aws iot list-targets-for-policy --policy-name "MyLightBulbPolicy"
```

Add the optional `--page-size` *`number`* parameter to specify the maximum number of results to be returned for each query, and the `--marker` *`string`* parameter on subsequent calls to retrieve the next set of results, if any.

The **ListTargetsForPolicy** command returns a list of targets and the token to use to retrieve more results:

```
{
    "nextMarker": "string",
    "targets": [ "string" ... ]
}
```

Get effective policies for a thing

You can use the **GetEffectivePolicies** command to list the policies in effect for a thing, including the policies attached to any groups the thing belongs to (whether the group is a direct parent or indirect ancestor):

```
$ aws iot get-effective-policies \
  --thing-name "MyLightBulb" \
  --principal "arn:aws:iot:us-east-1:123456789012:cert/
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847"
```

Use the `--principal` parameter to specify the ARN of the certificate attached to the thing. If you are using Amazon Cognito identity authentication, use the `--cognito-identity-pool-id` parameter and, optionally, add the `--principal` parameter to specify an Amazon Cognito identity. If you specify only the `--cognito-identity-pool-id`, the policies associated with that identity pool's role for unauthenticated users are returned. If you use both, the policies associated with that identity pool's role for authenticated users are returned.

The `--thing-name` parameter is optional and can be used instead of the `--principal` parameter. When used, the policies attached to any group the thing belongs to, and the policies attached to any parent groups of these groups (up to the root group in the hierarchy) are returned.

The **GetEffectivePolicies** command returns a list of policies:

```
{
    "effectivePolicies": [
        {
            "policyArn": "string",
            "policyDocument": "string",
            "policyName": "string"
        }
        ...
    ]
}
```

Test authorization for MQTT actions

You can use the **TestAuthorization** command to test whether an MQTT action (`Publish`, `Subscribe`) is allowed for a thing:

```
aws iot test-authorization \
    --principal "arn:aws:iot:us-east-1:123456789012:cert/
a0c01f5835079de0a7514643d68ef8414ab739a1e94ee4162977b02b12842847" \
    --auth-infos "{\"actionType\": \"PUBLISH\", \"resources\": [ \"arn:aws:iot:us-
east-1:123456789012:topic/my/topic\"]}"
```

Use the `--principal` parameter to specify the ARN of the certificate attached to the thing. If using Amazon Cognito Identity authentication, specify a Cognito Identity as the `--principal` or use the `--cognito-identity-pool-id` parameter, or both. (If you specify only the `--cognito-identity-pool-id` then the policies associated with that identity pool's role for unauthenticated users are considered. If you use both, the policies associated with that identity pool's role for authenticated users are considered.

Specify one or more MQTT actions you want to test by listing sets of resources and action types following the `--auth-infos` parameter. The `actionType` field should contain "PUBLISH", "SUBSCRIBE", "RECEIVE", or "CONNECT". The `resources` field should contain a list of resource ARNs. See AWS IoT Core policies for more information.

You can test the effects of adding policies by specifying them with the `--policy-names-to-add` parameter. Or you can test the effects of removing policies by them with the `--policy-names-to-skip` parameter.

You can use the optional `--client-id` parameter to further refine your results.

The **TestAuthorization** command returns details on actions that were allowed or denied for each set of `--auth-infos` queries you specified:

```
{
    "authResults": [
        {
            "allowed": {
                "policies": [
                    {
                        "policyArn": "string",
                        "policyName": "string"
                    }
                ]
            },
            "authDecision": "string",
            "authInfo": {
                "actionType": "string",
                "resources": [ "string" ]
            },
            "denied": {
                "explicitDeny": {
                    "policies": [
                        {
                            "policyArn": "string",
```

```
                              "policyName": "string"
                          }
                      ]
                  },
                  "implicitDeny": {
                      "policies": [
                          {
                              "policyArn": "string",
                              "policyName": "string"
                          }
                      ]
                  }
              },
              "missingContextValues": [ "string" ]
          }
      ]
}
```

Dynamic thing groups

Dynamic thing groups are created from specific search queries in the registry. Search query parameters such as device connectivity, device shadow creation, and AWS IoT Device Defender violations data support this. Dynamic thing groups require fleet indexing enabled to index, search, and aggregate your devices' data. You can preview the things in a dynamic thing group using a fleet indexing search query before creating it. For more information, see Fleet indexing and Query syntax.

Note

Dynamic thing group operations are metered under registry operations. For more information, see AWS IoT Core additional metering details.

Dynamic thing groups differ from static thing groups in the following ways:

- Thing membership is not explicitly defined. To create a dynamic thing group, define a search query string to determine group membership.
- Dynamic thing groups can't be part of a hierarchy.
- Dynamic thing groups can't have policies applied to them.

- You use a different set of commands to create, update, and delete dynamic thing groups. For all other operations, you use the same commands for both types of thing groups.
- The number of dynamic groups per AWS account is limited.

- Don't use personally identifiable information in your thing group name. The thing group name can appear in unencrypted communications and reports.

For more information about static thing groups, see Static thing groups.

Use cases of dynamic thing groups

You can use dynamic thing groups for the following use cases:

Specify a dynamic thing group as a target for a job

Creating a continuous job with a dynamic thing group as target allows you to automatically target devices when they meet the desired criteria. The criteria can be the connectivity state or any criteria stored in registry or shadow such as software version or model. If a thing doesn't appear in the dynamic thing group, it won't receive the job document from the job.

For example, if your device fleet requires a firmware update to minimize the risk of interruption during the update process, and you only want to update the firmware on devices with a battery life greater than 80%. You can create a dynamic thing group called 80PercentBatteryLife that only includes devices with a battery life above 80% and use it as the target for your job. Only devices that meet your battery life criteria will receive the firmware update. As devices reach the 80% battery life criteria, they are automatically added to the dynamic thing group and will receive the firmware update.

You may also have multiple device models with different firmware or operating system, necessitating different versions of new software updates. This is the most common use case for dynamic groups with continuous jobs, where you can create a dynamic group for each device model, firmware and OS combination. You can then set up continuous jobs to each of these dynamic groups to push software updates as devices automatically become members of these groups based on the defined criteria.

For more information about specifying thing groups as job targets, see CreateJob.

Use dynamic group membership changes to perform desired actions

Each time a device is added to or removed from a dynamic thing group, a notification is sent to an MQTT topic as part of registry event updates. You can configure AWS IoT Core rules to interact with AWS services based on the dynamic group membership updates and take desired actions. Example actions include writing to Amazon DynamoDB, invoking a Lambda function, or sending a notification to Amazon SNS.

Add devices to a dynamic thing group for automatic violation detection

AWS IoT Device Defender Detect customers can define a security profile on a dynamic thing group. Devices of the dynamic thing group are automatically detected for violations by the security profile defined on the group.

Set log levels on dynamic thing groups to observe devices with fine-grained logging

You can specify a log level on a dynamic thing group. This is useful if you only want to customize logging level and detail for devices that meet certain criteria. For example, if you suspect devices with certain firmware version are causing errors on a specific rule's published topic, you might want to set detailed logging to debug these issues. In this case, you can create a dynamic group for all devices that have this firmware version, which we assume is stored as a registry attribute or in a device shadow. You can then set a debug level, with logging target defined as this dynamic thing group. For more information about fine-grained logging, see Monitor AWS IoT using CloudWatch Logs. For more information about how to specify a logging level for a specific thing group, see Configure resource-specific logging in AWS IoT.

Create a dynamic thing group

Use the **CreateDynamicThingGroup** command to create a dynamic thing group. To create a dynamic thing group for the 80PercentBatteryLife scenario, use the **create-dynamic-thing-group** CLI command:

```
$ aws iot create-dynamic-thing-group --thing-group-name "80PercentBatteryLife" --query-
string "attributes.batterylife80"
```

Note

Don't use personally identifiable information in your dynamic thing group names.

The **CreateDynamicThingGroup** command returns a response. The response contains the index name, query string, query version, thing group name, thing group ID, and the Amazon Resource Name (ARN) of your thing group:

```
{
    "indexName": "AWS_Things",
    "queryVersion": "2017-09-30",
    "thingGroupName": "80PercentBatteryLife",
    "thingGroupArn": "arn:aws:iot:us-
west-2:123456789012:thinggroup/80PercentBatteryLife",
    "queryString": "attributes.batterylife80\n",
    "thingGroupId": "abcdefgh12345678ijklmnop12345678qrstuvwx"
}
```

The creation of dynamic thing groups doesn't happen at once. The dynamic thing group backfill takes time to complete. When you create a dynamic thing group, the status of the group is set to `BUILDING`. When the backfill is complete, the status changes to `ACTIVE`. To check the status of your dynamic thing group, use the DescribeThingGroup command.

Describe a dynamic thing group

Use the **DescribeThingGroup** command to get information about a dynamic thing group:

```
$ aws iot describe-thing-group --thing-group-name "80PercentBatteryLife"
```

The **DescribeThingGroup** command returns information about the specified group:

```
{
    "status": "ACTIVE",
    "indexName": "AWS_Things",
    "thingGroupName": "80PercentBatteryLife",
    "thingGroupArn": "arn:aws:iot:us-
west-2:123456789012:thinggroup/80PercentBatteryLife",
    "queryString": "attributes.batterylife80\n",
    "version": 1,
    "thingGroupMetadata": {
        "creationDate": 1548716921.289
    },
    "thingGroupProperties": {},
    "queryVersion": "2017-09-30",
    "thingGroupId": "84dd9b5b-2b98-4c65-84e4-be0e1ecf4fd8"
}
```

Running **DescribeThingGroup** on a dynamic thing group returns attributes that are specific to the dynamic thing groups. Example return attributes are the queryString and the status.

The status of a dynamic thing group can take the following values:

`ACTIVE`

The dynamic thing group is ready for use.

`BUILDING`

The dynamic thing group is being created, and thing membership is being processed.

`REBUILDING`

The dynamic thing group's membership is being updated, following the adjustment of the group's search query.

> **Note**
>
> After you create a dynamic thing group, use it regardless of its status. Only dynamic thing groups with an `ACTIVE` status include all of the things that match the search query for that dynamic thing group. Dynamic thing groups with `BUILDING` and `REBUILDING` statuses might not include all of the things that match the search query.

Update a dynamic thing group

Use the **UpdateDynamicThingGroup** command to update the attributes of a dynamic thing group, including the group's search query. The following command updates two attributes. One is the thing group description, and the other is the query string that changes the membership criteria to battery life > 85:

```
$ aws iot update-dynamic-thing-group --thing-group-name "80PercentBatteryLife" --thing-group-properties "thingGroupDescription=\"This thing group contains devices with a battery life greater than 85 percent.\"" --query-string "attributes.batterylife85"
```

The **UpdateDynamicThingGroup** command returns a response that contains the group's version number after the update:

```
{
    "version": 2
}
```

The update of a dynamic thing group doesn't happen at once. The dynamic thing group backfill takes time to complete. When you update a dynamic thing group, the status of the group changes to `REBUILDING` while the group updates its membership. When the backfill is complete, the status changes to `ACTIVE`. To check the status of your dynamic thing group, use the [DescribeThingGroup](https://docs.aws.amazon.com/iot/latest/apireference/API_DescribeThingGroup.html) command.

Delete a dynamic thing group

Use the **DeleteDynamicThingGroup** command to delete a dynamic thing group:

```
$ aws iot delete-dynamic-thing-group --thing-group-name "80PercentBatteryLife"
```

The **DeleteDynamicThingGroup** command doesn't produce any output.

Commands that show which groups a thing belongs to (for example, **ListGroupsForThing**) might continue to show the group while records in the cloud are being updated.

Dynamic and Static Thing Group Limitations

Dynamic thing groups and static thing groups share the following limitations:

- The number of attributes that a thing group can have is limited.
- The number of groups to which a thing can belong is limited.
- You can't rename thing groups.
- Thing group names can't contain international characters, such as û, é, and ñ.

Dynamic Thing Group Limitations

Dynamic thing groups have the following limitations:

Fleet indexing

With the fleet indexing service enabled, you can perform search queries on your fleet of devices. You can create and manage dynamic thing groups after the fleet indexing backfill is completed.

The completion time for the backfill process is directly affeted by the size of your device fleet registered with the AWS Cloud. After you enable the fleet indexing service for dynamic thing groups, you cannot disable it until you delete all of your dynamic thing groups.

> **Note**
>
> If you have permissions to query the fleet index, you can access the data of things across the entire fleet.

The number of dynamic thing groups is limited

The number of dynamic thing groups is limited.

Successful commands can log errors

When you create or update a dynamic thing group, it's possible that some things are eligible for inclusion in a dynamic thing group, but they are not added to it. That scenario will cause a succeessful create or update command while logging an error and generating an `AddThingToDynamicThingGroupsFailed` metric. A single metric can represent multiple log entries.

An error log entry in the CloudWatch log is created when the following occurs:

- An eligible thing can't be added to a dynamic thing group.
- A thing is removed from a dynamic thing group to add it to another group.

When a thing becomes eligible to be added to a dynamic thing group, consider the following:

- Is the thing already in as many groups as it can be? (See limits)
  - **NO:** The thing is added to the dynamic thing group.
  - **YES:** Is the thing a member of any dynamic thing groups?
    - **NO:** The thing can't be added to the dynamic thing group, an error is logged, and an `AddThingToDynamicThingGroupsFailed` metric is generated.
    - **YES:** Is the dynamic thing group to join older than any dynamic thing group that the thing is already a member of?
      - **NO:** The thing can't be added to the dynamic thing group, an error is logged, and an `AddThingToDynamicThingGroupsFailed` metric is generated.

        - **YES:** Remove the thing from the most recent dynamic thing group, log an error, and add the thing to the dynamic thing group. This generates an error and an `AddThingToDynamicThingGroupsFailed` metric for the dynamic thing group from which the thing was removed.

When a thing in a dynamic thing group no longer meets the search query, the thing is removed from the dynamic thing group. Likewise, when a thing is updated to meet a dynamic thing group's search query, the thing is then added to the group as previously described. These additions and removals are normal and don't produce error log entries.

With `overrideDynamicGroups` enabled, static groups take priority over dynamic groups

The number of groups to which a thing can belong is limited. When you use the AddThingToThingGroup or UpdateThingGroupsForThing commands to update thing membership, adding the `--overrideDynamicGroups` parameter gives static thing groups priority over dynamic thing groups.

When you add a thing to a static thing group, consider the following:

- Does the thing already belong to the maximum number of groups?
  - **NO:** The thing is added to the static thing group.
  - **YES:** Is the thing in any dynamic groups?
    - **NO:** The thing can't be added to the thing group. The command raises an exception.
    - **YES:** Was **--overrideDynamicGroups** enabled?
      - **NO:** The thing can't be added to the thing group. The command raises an exception.
      - **YES:** The thing is removed from the most recently created dynamic thing group, an error is logged, and an `AddThingToDynamicThingGroupsFailed` metric is generated for the dynamic thing group from which the thing was removed. Then, the thing is added to the static thing group.

Older dynamic thing groups take priority over newer ones

The number of groups to which a thing can belong is limited. When a create or update operation creates additional group eligibility for a thing and the thing has reached its group limit, removal from another dynamic thing group can occur to enable this addition. For more information about how this occurs, see Successful commands can log errors and With `overrideDynamicGroups` enabled, static groups take priority over dynamic groups for examples.

When a thing is removed from a dynamic thing group, an error is logged and an event is raised.

You can't apply policies to dynamic thing groups

Attempting to apply a policy to a dynamic thing group generates an exception.

Dynamic thing group membership is eventually consistent

Only the final state of a thing is evaluated for the registry. Intermediary states can be skipped if states are updated rapidly. Avoid associating a rule or job with a dynamic thing group whose membership depends on an intermediary state.

Associating an AWS IoT thing to an MQTT client connection

An exclusive thing association is when you attach an X.509 certificate to a single AWS IoT thing. In this case, the certificate cannot be used with other things. By ensuring that a certificate is used only by a single IoT thing, it helps prevent security vulnerabilities.

In AWS IoT, the client ID is a unique identifier for a thing or a device when it connects to the AWS IoT Core MQTT broker. If you use a non-exclusive association, multiple things can be attached to the same certificate. When non-exclusive thing association is in place, to maintain a clear association and to avoid potential conflicts, you must match your client ID with the thing name.

In this topic

- Use cases
- How to associate a thing to a connection

Use cases

Associating a thing to a connection provides the following capabilities.

> **Note**
>
> Note that if your IoT thing and client connection has a non-exclusive association, you can use all the following capabilities except the lifecycle events capability. To include your thing

name in the lifecycle event messages, you IoT thing and client connection must have an exclusive association.

Thing policy variables - You can use thing policy variables to authorize device access to AWS IoT API operations. These variables allow you to write AWS IoT Core policies that grant or deny permissions based on thing properties like names, types, and attribute values. By using thing policy variables, you can apply the same policy to control multiple AWS IoT Core devices. This allows you to simplify policy management and reduce resource duplication. For more information, see Thing policy variables.

Lifecycle events - You can receive the thing name in lifecycle events (for example, connect, disconnect and subscribe, and unsubscribe). This allows processing of the thing name included in the messages, such as in rules. For more information, see Lifecycle events.

Resource-specific logging - You can configure resource-specific logging for thing groups, and easily apply the desired logging configuration for all things within the thing group defined. For more information, see ???.

Cost allocation - You can create billing groups with custom tags for cost allocation and add the things to these groups. For more information, see Billing groups.

How to associate a thing to a connection

If your client ID matches your thing's name in the registry, after you attach an X.509 certificate to that IoT thing, AWS IoT Core will associate the client connection with the thing. If your client ID doesn't match the thing's name in the registry, you can exclusively attach an X.509 certificate to the thing to establish this association. The thing that has this exclusive attachment is called an exclusive thing. Otherwise, it's called a non-exclusive thing. When a certificate is associated with an exclusive thing, this certificate can only be associated with other things if you detach it from the exclusive thing. In this section, choose either AWS Management Console or AWS CLI to associate a thing to a connection.

AWS Management Console

To attach a certificate to a thing exclusively using the AWS Management Console.

1. Open the AWS IoT home page in the AWS IoT console. On the left navigation, from **Security**, choose **Certificates**.

2. On the **Certificates** page, choose a certificate you want to attach a thing to. Then choose **Attach to things** from **Actions** on the upper right corner of the page.

   Alternatively, choose a certificate and navigate to the certificate details page. Choose the **Things** tab, then choose **Attach to things**.

3. On the **Attach certificate to thing(s)** page, check the **Associate thing to connection** check box. Then choose a thing to attach this certificate to from the **Things** dropdown list.

4. Choose **Attach thing(s)**. If the action succeeds, you will see a banner that says "Successfully attached a thing to your certificate", and the thing will be added to the **Things** tab.

To detach a certificate from an exclusive thing using the AWS Management Console

1. Open the AWS IoT home page in the AWS IoT console. On the left navigation, from **Security**, choose **Certificates**.
2. On the **Certificates** page, choose a certificate and navigate to the certificate details page.
3. On the certificate details page, choose the **Things** tab. Then choose a thing that you want to detach the certificate to. Choose **Detach things**.
4. On the **Detach things** window, confirm your action. Choose **Detach**. If the action succeeds, you will see a banner that says "Successfully detached a thing from your certificate", and the thing will no longer appear in the **Things** tab.

AWS CLI

1. To attach a certificate to an thing using AWS CLI, run the **attach-thing-principal** command. To specify the exclusive certificate-to-thing attachment, you must specify `EXCLUSIVE_THING` in the `--thing-principal-type` field. An example command can be the following.

```
aws iot attach-thing-principal \
    --thing-name "thing_1" \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8" \
    --thing-principal-type "EXCLUSIVE_THING"
```

   This command doesn't produce any output. For more information, see ???.

2. To list the things associated with the specified certificate along with the attachment type, run the `list-principal-things-v2` command. The attachment type refers to how the certificate is attached to the thing. An example command can be the following.

```
$ aws iot list-principal-things-v2 \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
```

   The output can look like the following.

```
{
    "PrincipalThingObjects": [
        {
            "thingPrincipalType": "EXCLUSIVE_THING",
            "thing": "arn:aws:iot:us-east-1:123456789012:thing/thing_1"
        }
    ]
}
```

   For more information, see ???.

3. To list the principals associated with the specified thing along with the attachment type, run the `list-thing-principals-v2` command . The attachment type refers to how the certificate is attached to the thing. An example command can be the following.

```
$ aws iot list-thing-principals-v2 \
    --thing-name "thing_1"
```

   The output can look like the following.

```
{
    "ThingPrincipalObjects": [
        {
            "thingPrincipalType": "EXCLUSIVE_THING",
            "principal": "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8"
        }
    ]
```

```
}
```

For more information, see ???.

4. To detach a certificate from a thing, run the detach-thing-principal command.

```
aws iot detach-thing-principal \
    --principal "arn:aws:iot:us-
east-1:123456789012:cert/
2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8" \
    --thing-name "thing_1"
```

This command doesn't produce any output. For more information, see ???.

Adding propagating attributes for message enrichment

In AWS IoT Core, you can enrich MQTT messages from devices by adding propagating attributes, which are contextual metadata from thing attributes or connection details. This process, known as message enrichment, can be helpful in various scenarios. For example, you can enrich messages for every inbound publish operation without making any device side changes or needing to use rules. By leveraging propagating attributes, you can benefit from a more efficient and cost-effective way to enrich your IoT data without the complexities of configuring rules or managing republishing configurations.

The message enrichment feature is available to AWS IoT Core customers who use basic ingest and message broker. It's important to note while publishing devices can use any MQTT version, subscribers (applications or services consuming messages) must support MQTT 5 to receive the enriched messages with propagating attributes. The enriched messages will be added as MQTT 5 user properties to every message published from devices. If you use rules, you can leverage the get_user_properties function to retrieve the enriched data for message routing or processing based on the data.

In AWS IoT Core, you can add propagating attributes when you create or update a thing type, by using the AWS Management Console or the AWS CLI.

> **Important**
>
> When adding propagating attributes, you must make sure that the client publishing the message has been authenticated with a certificate. For more information, see Client authentication.

> **Note**
>
> If you attempt to test this feature using the MQTT test client within console, it may not work since this feature requires MQTT clients authenticated with an associated certificate.

AWS Management Console

To add propagating attributes for message enrichment using the AWS Management Console

1. Open the AWS IoT home page in the AWS IoT console. On the left navigation, from **Manage**, choose **All devices**. Then choose **Thing types**.
2. On the **Thing types** page, choose **Create thing type**.

   To configure message enrichment by updating a thing type, choose a thing type. Then on the thing type details page, choose **Update**.
3. On the **Create thing type** page, choose or enter the thing type information in **Thing type properties**.

   If you choose to update a thing type, you will see **Thing type properties** after you choose **Update** in the previous step.
4. In **Additional configuration**, expand **Propagating attributes**. Then choose **Thing attribute** and enter the thing attribute you want to populate to the published MQTT5 messages. Using the console, you can add up to three thing attributes.

   On the **Propagating attributes** section, choose **Connection attribute** and enter the attribute type and optionally the attribute name.
5. Optionally, add tags. Then choose **Create thing type**.

   If you choose to update a thing type, choose **Update thing type**.

AWS CLI

1. To add propagating attributes for message enrichment by creating a new thing type using the AWS CLI, run the **create-thing-type** command. An example command can be the following.

```
aws iot create-thing-type \
    --thing-type-name "LightBulb" \
    --thing-type-properties "{\"mqtt5Configuration\":{\"propagatingAttributes\":
[{\"userPropertyKey\":\"iot:ClientId\", \"connectionAttribute\":\"iot:ClientId\"},
 {\"userPropertyKey\":\"test\", \"thingAttribute\":\"A\"}]}}" \
```

   The output of the command can look like the following.

```
{
 "thingTypeName": "LightBulb",
 "thingTypeArn": "arn:aws:iot:us-west-2:123456789012:thingtype/LightBulb",
 "thingTypeId": "ce3573b0-0a3c-45a7-ac93-4e0ce14cd190"
}
```

2. To configure message enrichment by updating a thing type using AWS CLI, run the **update-thing-type** command. Note that you can only update `mqtt5Configuration` when you run this command. An example command can be the following.

```
aws iot update-thing-type \
    --thing-type-name "MyThingType" \
    --thing-type-properties "{\"mqtt5Configuration\":{\"propagatingAttributes\":
[{\"userPropertyKey\":\"iot:ClientId\", \"connectionAttribute\":\"iot:ClientId\"},
 {\"userPropertyKey\":\"test\", \"thingAttribute\":\"A\"}]}}" \
```

   This command doesn't produce any output.

3. To describe a thing type, run the `describe-thing-type` command. This command will produce an output with message enrichment configuration information in the `thing-type-properties` field. An example command can be the following.

```
aws iot describe-thing-type \
    --thing-type-name "LightBulb"
```

   The output can look like the following.

```
{
 "thingTypeName": "LightBulb",
 "thingTypeId": "bdf72512-0116-4392-8d79-bf39b17ef73d",
 "thingTypeArn": "arn:aws:iot:us-east-1:123456789012:thingtype/LightBulb",
 "thingTypeProperties": {
  "mqtt5Configuration": {
   "propagatingAttributes": [
    {
     "userPropertyKey": "iot:ClientId",
     "connectionAttribute": "iot:ClientId"
    },
    {
     "userPropertyKey": "test",
     "thingAttribute": "attribute"
    }
   ]
  }
 },
 "thingTypeMetadata": {
  "deprecated": false,
  "creationDate": "2024-10-18T17:37:46.656000+00:00"
 }
}
```


For more information, see ???.

Tagging your AWS IoT resources

To help you manage and organize your thing groups, thing types, topic rules, jobs, scheduled audits and security profiles you can optionally assign your own metadata to each of these resources in the form of tags. This section describes tags and shows you how to create them.

To help you manage your costs related to things, you can create billing groups that contain things. You can then assign tags that contain your metadata to each of these billing groups. This section also discusses billing groups and the commands available to create and manage them.

Tag basics

You can use tags to categorize your AWS IoT resources in different ways (for example, by purpose, owner, or environment). This is useful when you have many resources of the same type — you can quickly identify a resource based on the tags you've assigned to it. Each tag consists of a key and optional value, both of which you define. For example, you can define a set of tags for your thing types that helps you track devices by type. We recommend that you create a set of tag keys that meets your needs for each kind of resource. Using a consistent set of tag keys makes it easier for you to manage your resources.

You can search for and filter resources based on the tags you add or apply. You can also use billing group tags to categorize and track your costs. You can also use tags to control access to your resources as described in Using tags with IAM policies.

For ease of use, the Tag Editor in the AWS Management Console provides a central, unified way to create and manage your tags. For more information, see Working with Tag Editor in Working with the AWS Management Console.

You can also work with tags using the AWS CLI and the AWS IoT API. You can associate tags with thing groups, thing types, topic rules, jobs, security profiles, policies, billing groups, and the packages and versions associated with things when you create them by using the `Tags` field in the following commands:

- CreateBillingGroup
- CreateDestination
- CreateDeviceProfile

- CreateDynamicThingGroup
- CreateJob
- CreateOTAUpdate
- CreatePolicy
- CreateScheduledAudit
- CreateSecurityProfile
- CreateServiceProfile
- CreateStream
- CreateThingGroup
- CreateThingType
- CreateTopicRule
- CreateWirelessGateway
- CreateWirelessDevice

You can add, modify, or delete tags for existing resources that support tagging by using the following commands:

- TagResource
- ListTagsForResource
- UntagResource

You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags associated with the resource are also deleted.

Tag restrictions and limitations

The following basic restrictions apply to tags:

- Maximum number of tags per resource — 50
- Maximum key length — 127 Unicode characters in UTF-8
- Maximum value length — 255 Unicode characters in UTF-8

- Tag keys and values are case sensitive.
- Do not use the `aws:` prefix in your tag names or values. It's reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix don't count against your tags per resource limit.
- If your tagging schema is used across multiple services and resources, remember that other services might have restrictions on allowed characters. Allowed characters include letters, spaces, and numbers representable in UTF-8, and the following special characters: + - = . _ : / @.

Using tags with IAM policies

You can apply tag-based resource-level permissions in the IAM policies you use for AWS IoT API actions. This gives you better control over what resources a user can create, modify, or use. You use the `Condition` element (also called the `Condition` block) with the following condition context keys and values in an IAM policy to control user access (permissions) based on a resource's tags:

- Use `aws:ResourceTag/tag-key: tag-value` to allow or deny user actions on resources with specific tags.
- Use `aws:RequestTag/tag-key: tag-value` to require that a specific tag be used (or not used) when making an API request to create or modify a resource that allows tags.
- Use `aws:TagKeys: [tag-key, ...]` to require that a specific set of tag keys be used (or not used) when making an API request to create or modify a resource that allows tags.

Note

The condition context keys and values in an IAM policy apply only to those AWS IoT actions where an identifier for a resource capable of being tagged is a required parameter. For example, the use of DescribeEndpoint is not allowed or denied on the basis of condition context keys and values because no taggable resource (thing groups, thing types, topic rules, jobs, or security profile) is referenced in this request. For more information about AWS IoT resources that are taggable and condition keys they support, read Actions, resources, and condition keys for AWS IoT.

For more information about using tags, see Controlling Access Using Tags in the *AWS Identity and Access Management User Guide*. The IAM JSON Policy Reference section of that guide has detailed syntax, descriptions, and examples of the elements, variables, and evaluation logic of JSON policies in IAM.

The following example policy applies two tag-based restrictions for the `ThingGroup` actions. An IAM user restricted by this policy:

- Can't create a thing group the tag "env=prod" (in the example, see the line `"aws:RequestTag/env" : "prod"`).
- Can't modify or access a thing group that has an existing tag "env=prod" (in the example, see the line `"aws:ResourceTag/env" : "prod"`).

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "iot:CreateThingGroup",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/env": "prod"
        }
      }
    },
    {
      "Effect": "Deny",
      "Action": [
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:UpdateThingGroup"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/env": "prod"
        }
      }
    },
    {
      "Effect": "Allow",
```

```
        "Action": [
          "iot:CreateThingGroup",
          "iot:DeleteThingGroup",
          "iot:DescribeThingGroup",
          "iot:UpdateThingGroup"
        ],
        "Resource": "*"
      }
    ]
}
```

You can also specify multiple tag values for a given tag key by enclosing them in a list, like this:

```
            "StringEquals" : {
              "aws:ResourceTag/env" : ["dev", "test"]
            }
```

Note

If you allow or deny users access to resources based on tags, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, it's possible for a user to circumvent your restrictions and gain access to a resource by modifying its tags.

Billing groups

AWS IoT doesn't allow you to directly apply tags to individual things, but it does allow you to place things in billing groups and to apply tags to these. For AWS IoT, allocation of cost and usage data based on tags is limited to billing groups.

AWS IoT Core for LoRaWAN resources, such as wireless devices and gateways, can't be added to billing groups. However, they can be associated with AWS IoT things, which can be added to billing groups.

The following commands are available:

- AddThingToBillingGroup adds a thing to a billing group.

- CreateBillingGroup creates a billing group.
- DeleteBillingGroup deletes the billing group.
- DescribeBillingGroup returns information about a billing group.
- ListBillingGroups lists the billing groups you have created.
- ListThingsInBillingGroup lists the things you have added to the given billing group.
- RemoveThingFromBillingGroup removes the given thing from the billing group.
- UpdateBillingGroup updates information about the billing group.
- CreateThing allows you to specify a billing group for the thing when you create it.
- DescribeThing returns the description of a thing including the billing group the thing belongs to, if any.

The AWS IoT Wireless API provides these actions to associate wireless devices and gateways with AWS IoT things.

- AssociateWirelessDeviceWithThing
- AssociateWirelessGatewayWithThing

Viewing cost allocation and usage data

You can use billing group tags to categorize and track your costs. When you apply tags to billing groups (and so to the things they include), AWS generates a cost allocation report as a comma-separated value (CSV) file with your usage and costs aggregated by your tags. You can apply tags that represent business categories (such as cost centers, application names, or owners) to organize your costs across multiple services. For more information about using tags for cost allocation, see Use Cost Allocation Tags in the AWS Billing and Cost Management User Guide.

> **Note**
>
> To accurately associate usage and cost data with those things you have placed in billing groups, each device or application must:
>
> - Be registered as a thing in AWS IoT. For more information, see Managing devices with AWS IoT.
> - Connect to the AWS IoT message broker through MQTT using only the thing's name as the client ID. For more information, see the section called “Device communication

protocols”. If you client ID doesn't match the thing name, you can enable the exclusive thing attachment to establish the association. For more information, see ???.

- Authenticate using a client certificate associated with the thing.

The following pricing dimensions are available for billing groups (based on the activity of things associated with the billing group):

- Connectivity (based on the thing name used as the client ID to connect).
- Messaging (based on messages inbound from, and outbound to, a thing; MQTT only).
- Shadow operations (based on the thing whose message triggered a shadow update).
- Rules triggered (based on the thing whose inbound message triggered the rule; does not apply to those rules triggered by MQTT lifecycle events).
- Thing index updates (based on the thing that was added to the index).
- Remote actions (based on the thing updated).
- AWS IoT Device Defender detect reports (based on the thing whose activity is reported).

Cost and usage data based on tags (and reported for a billing group) doesn't reflect the following activities:

- Device registry operations (including updates to things, thing groups, and thing types). For more information, see Managing devices with AWS IoT).
- Thing group index updates (when adding a thing group).
- Index search queries.
- Device provisioning.
- AWS IoT Device Defender audit reports.

Security in AWS IoT

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to AWS IoT, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

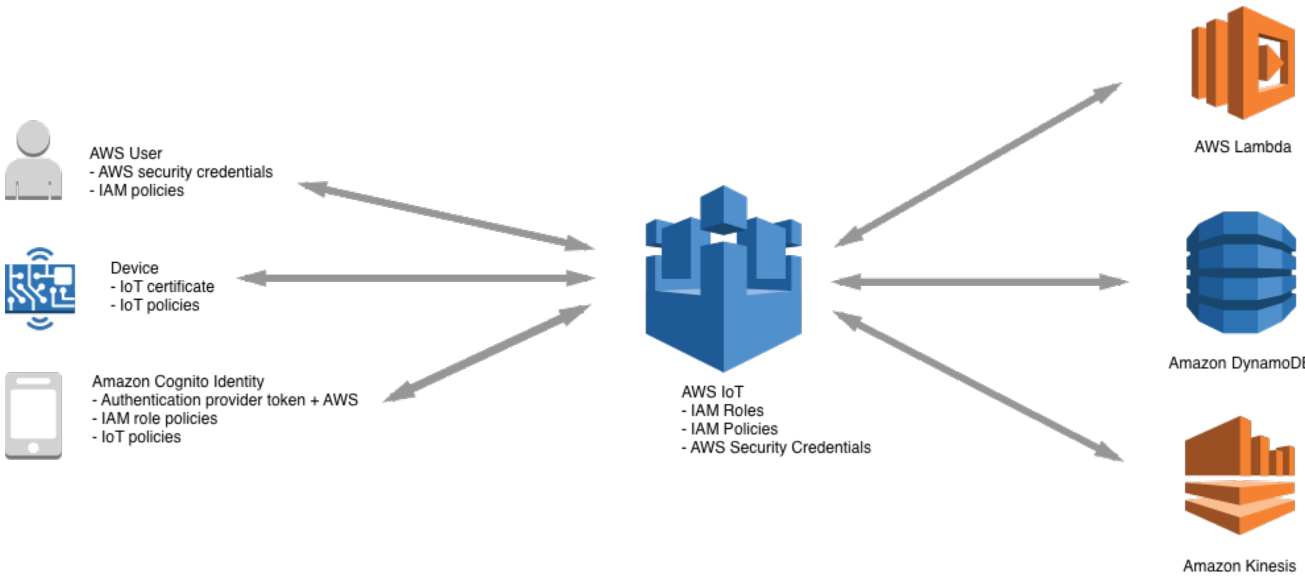
This documentation helps you understand how to apply the shared responsibility model when using AWS IoT. The following topics show you how to configure AWS IoT to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS IoT resources.

Topics

- Security monitoring of production fleets or devices with AWS IoT Core
- Security best practices in AWS IoT Core
- AWS training and certification

AWS IoT security

Each connected device or client must have a credential to interact with AWS IoT. All traffic to and from AWS IoT is sent securely over Transport Layer Security (TLS). AWS cloud security mechanisms protect data as it moves between AWS IoT and other AWS services.



- You are responsible for managing device credentials (X.509 certificates, AWS credentials, Amazon Cognito identities, federated identities, or custom authentication tokens) and policies in AWS IoT. You are also responsible for assigning unique identities to each device and managing the permissions for each device or group of devices.
- Your devices connect to AWS IoT using X.509 certificates or Amazon Cognito identities over a secure TLS connection. During research and development, and for some applications that make API calls or use WebSockets, you can also authenticate using IAM users and groups or custom authentication tokens. For more information, see IAM users, groups, and roles.
- When using AWS IoT authentication, the message broker is responsible for authenticating your devices, securely ingesting device data, and granting or denying access permissions you specify for your devices using AWS IoT policies.

- When using custom authentication, a custom authorizer is responsible for authenticating your devices and granting or denying access permissions you specify for your devices using AWS IoT or IAM policies.
- The AWS IoT rules engine forwards device data to other devices or other AWS services according to rules you define. It uses AWS Identity and Access Management to securely transfer data to its final destination. For more information, see Identity and access management for AWS IoT.

Authentication

Authentication is a mechanism where you verify the identity of a client or a server. Server authentication is the process where devices or other clients ensure they are communicating with an actual AWS IoT endpoint. Client authentication is the process where devices or other clients authenticate themselves with AWS IoT.

X.509 Certificate overview

X.509 certificates are digital certificates that use the X.509 public key infrastructure standard to associate a public key with an identity contained in a certificate. X.509 certificates are issued by a trusted entity called a certification authority (CA). The CA maintains one or more special certificates called CA certificates that it uses to issue X.509 certificates. Only the certification authority has access to CA certificates. X.509 certificate chains are used both for server authentication by clients and client authentication by the server.

Server authentication

When your device or other client attempts to connect to AWS IoT Core, the AWS IoT Core server will send an X.509 certificate that your device uses to authenticate the server. Authentication takes place at the TLS layer through validation of the X.509 certificate chain. This is the same method used by your browser when you visit an HTTPS URL. If you want to use certificates from your own certificate authority, see Manage your CA certificates.

When your devices or other clients establish a TLS connection to an AWS IoT Core endpoint, AWS IoT Core presents a certificate chain that the devices use to verify that they're communicating with AWS IoT Core and not another server impersonating AWS IoT Core. The chain that is presented depends on a combination of the type of endpoint the device is connecting to and the cipher suite that the client and AWS IoT Core negotiated during the TLS handshake.

Endpoint types

AWS IoT Core supports `iot:Data-ATS`. `iot:Data-ATS` endpoints present a server certificate signed by an Amazon Trust Services CA.

Certificates presented by ATS endpoints are cross signed by Starfield. Some TLS client implementations require validation of the root of trust and require that the Starfield CA certificates are installed in the client's trust stores.

> **Warning**
>
> Using a method of certificate pinning that hashes the whole certificate (including the issuer name, and so on) is not recommended because this will cause certificate verification to fail because the ATS certificates we provide are cross signed by Starfield and have a different issuer name.

> **Important**
>
> Use `iot:Data-ATS` endpoints. Symantec and Verisign certificates have been deprecated and are no longer supported by AWS IoT Core.

You can use the `describe-endpoint` command to create your ATS endpoint.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

The `describe-endpoint` command returns an endpoint in the following format.

```
account-specific-prefix.iot.your-region.amazonaws.com
```

> **Note**
>
> The first time `describe-endpoint` is called, an endpoint is created. All subsequent calls to `describe-endpoint` return the same endpoint.

> **Note**
>
> To see your `iot:Data-ATS` endpoint in the AWS IoT Core console, choose **Settings**. The console displays only the `iot:Data-ATS` endpoint.

Creating an `IotDataPlaneClient` with the AWS SDK for Java

To create an `IotDataPlaneClient` that uses an `iot:Data-ATS` endpoint, you must do the following.

- Create an `iot:Data-ATS` endpoint by using the DescribeEndpoint API.
- Specify that endpoint when you create the `IotDataPlaneClient`.

The following example performs both of these operations.

```
public void setup() throws Exception {
        IotClient client =
 IotClient.builder().credentialsProvider(CREDENTIALS_PROVIDER_CHAIN).region(Region.US_EAST_1).b
        String endpoint = client.describeEndpoint(r -> r.endpointType("iot:Data-
ATS")).endpointAddress();
        iot = IotDataPlaneClient.builder()
                                .credentialsProvider(CREDENTIALS_PROVIDER_CHAIN)
                                .endpointOverride(URI.create("https://" + endpoint))
                                .region(Region.US_EAST_1)
                                .build();
}
```

CA certificates for server authentication

Depending on which type of data endpoint you are using and which cipher suite you have negotiated, AWS IoT Core server authentication certificates are signed by one of the following root CA certificates:

Amazon Trust Services Endpoints (preferred)

> **Note**
>
> You might need to right click these links and select **Save link as...** to save these certificates as files.

- RSA 2048 bit key: Amazon Root CA 1.
- RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
- ECC 256 bit key: Amazon Root CA 3.
- ECC 384 bit key: Amazon Root CA 4. Reserved for future use.

These certificates are all cross-signed by the Starfield Root CA Certificate. All new AWS IoT Core regions, beginning with the May 9, 2018 launch of AWS IoT Core in the Asia Pacific (Mumbai) Region, serve only ATS certificates.

VeriSign Endpoints (legacy)

- RSA 2048 bit key: VeriSign Class 3 Public Primary G5 root CA certificate

Server authentication guidelines

There are many variables that can affect a device's ability to validate the AWS IoT Core server authentication certificate. For example, devices may be too memory constrained to hold all possible root CA certificates, or devices may implement a non-standard method of certificate validation. For these reasons we suggest following these guidelines:

- We recommend that you use your ATS endpoint and install all supported Amazon Root CA certificates.
- If you cannot store all of these certificates on your device and if your devices do not use ECC-based validation, you can omit the Amazon Root CA 3 and Amazon Root CA 4 ECC certificates. If your devices do not implement RSA-based certificate validation, you can omit the Amazon Root CA 1 and Amazon Root CA 2 RSA certificates. You might need to right click these links and select **Save link as...** to save these certificates as files.
- If you are experiencing server certificate validation issues when connecting to your ATS endpoint, try adding the relevant cross-signed Amazon Root CA certificate to your trust store. You might need to right click these links and select **Save link as...** to save these certificates as files.

  - Cross-signed Amazon Root CA 1
  - Cross-signed Amazon Root CA 2 - Reserved for future use.
  - Cross-signed Amazon Root CA 3
  - Cross-signed Amazon Root CA 4 - Reserved for future use.
- If you are experiencing server certificate validation issues, your device may need to explicitly trust the root CA. Try adding the Starfield Root CA Certificate to your trust store.
- If you still experience issues after executing the steps above, please contact AWS Developer Support.

> **Note**
>
> CA certificates have an expiration date after which they cannot be used to validate a server's certificate. CA certificates might have to be replaced before their expiration date. Make sure that you can update the root CA certificates on all of your devices or clients to help ensure ongoing connectivity and to keep up to date with security best practices.

> **Note**
>
> When connecting to AWS IoT Core in your device code, pass the certificate into the API you are using to connect. The API you use will vary by SDK. For more information, see the AWS IoT Core Device SDKs.

Client authentication

AWS IoT supports three types of identity principals for device or client authentication:

- X.509 client certificates
- IAM users, groups, and roles
- Amazon Cognito identities

These identities can be used with devices, mobile, web, or desktop applications. They can even be used by a user typing AWS IoT command line interface (CLI) commands. Typically, AWS IoT devices use X.509 certificates, while mobile applications use Amazon Cognito identities. Web and desktop applications use IAM or federated identities. AWS CLI commands use IAM. For more information about IAM identities, see Identity and access management for AWS IoT.

X.509 client certificates

X.509 certificates provide AWS IoT with the ability to authenticate client and device connections. Client certificates must be registered with AWS IoT before a client can communicate with AWS IoT. A client certificate can be registered in multiple AWS accounts in the same AWS Region to facilitate moving devices between your AWS accounts in the same region. See Using X.509 client certificates in multiple AWS accounts with multi-account registration for more information.

We recommend that each device or client be given a unique certificate to enable fine-grained client management actions, including certificate revocation. Devices and clients must also support rotation and replacement of certificates to help ensure smooth operation as certificates expire.

For information about using X.509 certificates to support more than a few devices, see Device provisioning to review the different certificate management and provisioning options that AWS IoT supports.

AWS IoT supports these types of X.509 client certificates:

- X.509 certificates generated by AWS IoT
- X.509 certificates signed by a CA registered with AWS IoT.
- X.509 certificates signed by a CA that is not registered with AWS IoT.

This section describes how to manage X.509 certificates in AWS IoT. You can use the AWS IoT console or AWS CLI to perform these certificate operations:

- Create AWS IoT client certificates
- Create your own client certificates
- Register a client certificate
- Activate or deactivate a client certificate
- Revoke a client certificate

For more information about the AWS CLI commands that perform these operations, see AWS IoT CLI Reference.

Using X.509 client certificates

X.509 certificates authenticate client and device connections to AWS IoT. X.509 certificates provide several benefits over other identification and authentication mechanisms. X.509 certificates enable asymmetric keys to be used with devices. For example, you could burn private keys into secure storage on a device so that sensitive cryptographic material never leaves the device. X.509 certificates provide stronger client authentication over other schemes, such as user name and password or bearer tokens, because the private key never leaves the device.

AWS IoT authenticates client certificates using the TLS protocol's client authentication mode. TLS support is available in many programming languages and operating systems and is commonly used for encrypting data. In TLS client authentication, AWS IoT requests an X.509 client certificate and validates the certificate's status and AWS account against a registry of certificates. It then challenges the client for proof of ownership of the private key that corresponds to the public key contained in the certificate. AWS IoT requires clients to send the Server Name Indication (SNI) extension to the Transport Layer Security (TLS) protocol. For more information on configuring the SNI extension, see Transport security in AWS IoT Core.

To facilitate a secure and consistent client connection to AWS IoT core, a X.509 client certificate must possess the following:

- Registered in AWS IoT Core. For more information, see Register a client certificate.
- Have a status state of `ACTIVE`. For more information, see Activate or deactivate a client certificate.
- Not yet reached the certificate expiration date.

You can create client certificates that use the Amazon Root CA and you can use your own client certificates signed by another certificate authority (CA). For more information about using the AWS IoT console to create certificates that use the Amazon Root CA, see Create AWS IoT client certificates. For more information about using your own X.509 certificates, see Create your own client certificates.

The date and time when certificates signed by a CA certificate expire are set when the certificate is created. X.509 certificates generated by AWS IoT expire at midnight UTC on December 31, 2049 (2049-12-31T23:59:59Z).

AWS IoT Device Defender can perform audits on your AWS account and devices supporting common IoT security best practices. This includes managing the expiration dates of X.509

certificates signed by your CA or the Amazon Root CA. For more information on managing a certificate's expiration date, see Device certificate expiring and CA certificate expiring.

On the official AWS IoT blog, a deeper dive into the management of device certificate rotation and security best practices is explored in How to manage IoT device certificate rotation using AWS IoT.

Using X.509 client certificates in multiple AWS accounts with multi-account registration

Multi-account registration makes it possible to move devices between your AWS accounts in the same Region or in different Regions. You can register, test, and configure a device in a pre-production account, and then register and use the same device and device certificate in a production account. You can also register the client certificate on the device or the device certificates without a CA that is registered with AWS IoT. For more information, see Register a client certificate signed by an unregistered CA (CLI).

> **Note**
>
> Certificates used for multi-account registration are supported on the `iot:Data-ATS`, `iot:Data` (legacy), `iot:Jobs`, and `iot:CredentialProvider` endpoint types. For more information about AWS IoT device endpoints, see AWS IoT device data and service endpoints.

Devices that use multi-account registration must send the Server Name Indication (SNI) extension to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field, when they connect to AWS IoT. AWS IoT uses the endpoint address in `host_name` to route the connection to the correct AWS IoT account. Existing devices that don't send a valid endpoint address in `host_name` will continue to work, but they will not be able to use the features that require this information. For more information about the SNI extension and to learn how to identify the endpoint address for the `host_name` field, see Transport security in AWS IoT Core.

To use multi-account registration

1. You can register the device certificates with a CA. You can register the signing CA in multiple accounts in `SNI_ONLY` mode and use that CA to register the same client certificate to multiple accounts. For more information, see Register a CA certificate in SNI_ONLY mode (CLI) - Recommended.

2. You can register the device certificates without a CA. See Register a client certificate signed by an unregistered CA (CLI). Registering a CA is optional. You're not required to register the CA that signed the device certificates with AWS IoT.

Certificate signing algorithms supported by AWS IoT

AWS IoT supports the following certificate-signing algorithms:

- SHA256WITHRSA
- SHA384WITHRSA
- SHA512WITHRSA
- SHA256WITHRSAANDMGF1 (RSASSA-PSS)
- SHA384WITHRSAANDMGF1 (RSASSA-PSS)
- SHA512WITHRSAANDMGF1 (RSASSA-PSS)
- DSA_WITH_SHA256
- ECDSA-WITH-SHA256
- ECDSA-WITH-SHA384
- ECDSA-WITH-SHA512

For more information about certificate authentication and security, see Device certificate key quality.

Note

The certificate signing request (CSR) must include a public key. The key can be either an RSA key with a length of at least 2,048 bits or an ECC key from NIST P-256, NIST P-384, or NIST P-521 curves. For more information, see CreateCertificateFromCsr in the *AWS IoT API Reference Guide*.

Key algorithms supported by AWS IoT

The table below shows how key algorithms are supported:

Key algorithm	Certificate signing algorithm	TLS version	Supported? Yes or No
RSA with a key size of at least 2048 bits	All	TLS 1.2 TLS 1.3	Yes
ECC NIST P-256/P-384/ P-521	All	TLS 1.2 TLS 1.3	Yes
RSA-PSS with a key size of at least 2048 bits	All	TLS 1.2	No
RSA-PSS with a key size of at least 2048 bits	All	TLS 1.3	Yes

To create a certificate using CreateCertificateFromCSR, you can use a supported key algorithm to generate a public key for your CSR. To register your own certificate using RegisterCertificate or RegisterCertificateWithoutCA, you can use a supported key algorithm to generate a public key for the certificate.

For more information, see Security policies.

Create AWS IoT client certificates

AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

This topic describes how to create a client certificate signed by the Amazon Root certificate authority and download the certificate files. After you create the client certificate files, you must install them on the client.

> **Note**
>
> Each X.509 client certificate provided by AWS IoT holds issuer and subject attributes that you set at the time of certificate creation. The certificate attributes are immutable only after the certificate is created.

You can use the AWS IoT console or the AWS CLI to create an AWS IoT certificate signed by the Amazon Root certificate authority.

Create an AWS IoT certificate (console)

To create an AWS IoT certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the navigation pane, choose **Security**, then choose **Certificates**, and then choose **Create**.
3. Choose **One-click certificate creation (recommended) - Create certificate**.
4. From the **Certificate created** page, download the client certificate files for the thing, public key, and private key to a secure location. These certificates generated by AWS IoT are only available for use with AWS IoT services.

   If you also need the Amazon Root CA certificate file, this page also has the link to the page where you can download it.
5. A client certificate has now been created and registered with AWS IoT. You must activate the certificate before you use it in a client.

   To activate the client certificate now, choose **Activate**. If you don't want to activate the certificate now, see Activate a client certificate (console) to learn how to activate the certificate later.

6. If you want to attach a policy to the certificate, choose **Attach a policy**.

   If you don't want to attach a policy now, choose **Done** to finish. You can attach a policy later.

After you complete the procedure, install the certificate files on the client.

Create an AWS IoT certificate (CLI)

The AWS CLI provides the **create-keys-and-certificate** command to create client certificates signed by the Amazon Root certificate authority. This command, however, does not download the Amazon Root CA certificate file. You can download the Amazon Root CA certificate file from CA certificates for server authentication.

This command creates private key, public key, and X.509 certificate files and registers and activates the certificate with AWS IoT.

```
aws iot create-keys-and-certificate \
    --set-as-active \
    --certificate-pem-outfile certificate_filename.pem \
```

```
    --public-key-outfile public_filename.key \
    --private-key-outfile private_filename.key
```

If you don't want to activate the certificate when you create and register it, this command creates private key, public key, and X.509 certificate files and registers the certificate, but it does not activate it. Activate a client certificate (CLI) describes how to activate the certificate later.

```
aws iot create-keys-and-certificate \
    --no-set-as-active \
    --certificate-pem-outfile certificate_filename.pem \
    --public-key-outfile public_filename.key \
    --private-key-outfile private_filename.key
```

Install the certificate files on the client.

Create your own client certificates

AWS IoT supports client certificates signed by any root or intermediate certificate authorities (CA). AWS IoT uses CA certificates to verify the ownership of certificates. To use device certificates signed by a CA that’s not Amazon’s CA, the CA’s certificate must be registered with AWS IoT so that we can verify the device certificate’s ownership.

AWS IoT supports multiple ways for bringing your own certificates (BYOC):

- First, register the CA that’s used for signing the client certificates and then register individual client certificates. If you want to register the device or client to its client certificate when it first connects to AWS IoT (also known as Just-in-Time Provisioning), you must register the signing CA with AWS IoT and activate auto-registration.
- If you can’t register the signing CA, you can choose to register client certificates without CA. For devices registered without CA, you’ll need to present Server Name Indication (SNI) when you connect them to AWS IoT.

Note

To register client certificates using CA, you must register the signing CA with AWS IoT, not any other CAs in the hierarchy.

> **Note**
>
> A CA certificate can be registered in `DEFAULT` mode by only one account in a Region. A CA certificate can be registered in `SNI_ONLY` mode by multiple accounts in a Region.

For more information about using X.509 certificates to support more than a few devices, see [Device provisioning](#) to review the different certificate management and provisioning options that AWS IoT supports.

Topics

- [Manage your CA certificates](#)
- [Create a client certificate using your CA certificate](#)

Manage your CA certificates

This section describes common tasks for managing your own certificate authority (CA) certificates.

You can register your certificate authority (CA) with AWS IoT if you are using client certificates signed by a CA that AWS IoT doesn't recognize.

If you want clients to automatically register their client certificates with AWS IoT when they first connect, the CA that signed the client certificates must be registered with AWS IoT. Otherwise, you don't need to register the CA certificate that signed the client certificates.

> **Note**
>
> A CA certificate can be registered in `DEFAULT` mode by only one account in a Region. A CA certificate can be registered in `SNI_ONLY` mode by multiple accounts in a Region.

Topics:

- [Create a CA certificate](#)
- [Register your CA certificate](#)
- [Deactivate a CA certificate](#)

Create a CA certificate

If you do not have a CA certificate, you can use OpenSSL v1.1.1i tools to create one.

> **Note**
>
> You can't perform this procedure in the AWS IoT console.

To create a CA certificate using OpenSSL v1.1.1i tools

1. Generate a key pair.

```
openssl genrsa -out root_CA_key_filename.key 2048
```

2. Use the private key from the key pair to generate a CA certificate.

```
openssl req -x509 -new -nodes \
    -key root_CA_key_filename.key \
    -sha256 -days 1024 \
    -out root_CA_cert_filename.pem
```

Register your CA certificate

These procedures describe how to register a certificate from a certificate authority (CA) that's not Amazon's CA. AWS IoT Core uses CA certificates to verify the ownership of certificates. To use device certificates signed by a CA that's not Amazon's CA, you must register the CA certificate with AWS IoT Core so that it can verify the device certificate's ownership.

Register a CA certificate (console)

> **Note**
>
> To register a CA certificate in the console, start in the console at Register CA certificate. You can register your CA in Multi-account mode and without the need to provide a verification certificate or access to the private key. A CA can be registered in Multi-account mode by multiple AWS accounts in the same AWS Region. You can register your CA in Single-account mode by providing a verification certificate and proof of ownership of CA's private key.

Register a CA certificate (CLI)

You can register a CA certificate in DEFAULT mode or SNI_ONLY mode. A CA can be registered in DEFAULT mode by one AWS account in one AWS Region. A CA can be registered in SNI_ONLY mode by multiple AWS accounts in the same AWS Region. For more information about CA certificate mode, see certificateMode.

> **Note**
>
> We recommend that you register a CA in SNI_ONLY mode. You don't need to provide a verification certificate or access to the private key, and you can register the CA by multiple AWS accounts in the same AWS Region.

Register a CA certificate in SNI_ONLY mode (CLI) - Recommended

Prerequisites

Make sure you have the following available on your computer before you continue:

- The root CA's certificate file (referenced in the following example as *root_CA_cert_filename.pem*)
- OpenSSL v1.1.1i or later

To register a CA certificate in SNI_ONLY mode using the AWS CLI

1. Register the CA certificate with AWS IoT. Using the **register-ca-certificate** command, enter the CA certificate file name. For more information, see register-ca-certificate in the *AWS CLI Command Reference*.

   ```
   aws iot register-ca-certificate \
       --ca-certificate file://root_CA_cert_filename.pem \
       --certificate-mode SNI_ONLY
   ```

   If successful, this command returns the *certificateId*.
2. At this point, the CA certificate has been registered with AWS IoT but is inactive. The CA certificate must be active before you can register any client certificates that it has signed.

   This step activates the CA certificate.

To activate the CA certificate, use the **update-certificate** command as follows. For more information, see update-certificate in the *AWS CLI Command Reference*.

```
aws iot update-ca-certificate \
    --certificate-id certificateId \
    --new-status ACTIVE
```

To see the status of the CA certificate, use the **describe-ca-certificate** command. For more information, see describe-ca-certificate in the *AWS CLI Command Reference*.

Register a CA certificate in `DEFAULT` mode (CLI)

Prerequisites

Make sure you have the following available on your computer before you continue:

- The root CA's certificate file (referenced in the following example as *`root_CA_cert_filename.pem`*)
- The root CA certificate's private key file (referenced in the following example as *`root_CA_key_filename.key`*)
- OpenSSL v1.1.1i or later

To register a CA certificate in `DEFAULT` mode using the AWS CLI

1. To get a registration code from AWS IoT, use **get-registration-code**. Save the returned `registrationCode` to use as the `Common Name` of the private key verification certificate. For more information, see get-registration-code in the *AWS CLI Command Reference*.

   ```
   aws iot get-registration-code
   ```

2. Generate a key pair for the private key verification certificate:

   ```
   openssl genrsa -out verification_cert_key_filename.key 2048
   ```

3. Create a certificate signing request (CSR) for the private key verification certificate. Set the `Common Name` field of the certificate to the `registrationCode` returned by **get-registration-code**.

```
openssl req -new \
    -key verification_cert_key_filename.key \
    -out verification_cert_csr_filename.csr
```

You are prompted for some information, including the `Common Name` for the certificate.

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
    State or Province Name (full name) []:
    Locality Name (for example, city) []:
    Organization Name (for example, company) []:
    Organizational Unit Name (for example, section) []:
    Common Name (e.g. server FQDN or YOUR name) []:your_registration_code
    Email Address []:

    Please enter the following 'extra' attributes
    to be sent with your certificate request
    A challenge password []:
    An optional company name []:
```

4. Use the CSR to create a private key verification certificate:

```
openssl x509 -req \
    -in verification_cert_csr_filename.csr \
    -CA root_CA_cert_filename.pem \
    -CAkey root_CA_key_filename.key \
    -CAcreateserial \
    -out verification_cert_filename.pem \
    -days 500 -sha256
```

5. Register the CA certificate with AWS IoT. Pass in the CA certificate file name and the private key verification certificate file name to the **register-ca-certificate** command, as follows. For more information, see [register-ca-certificate](https://docs.aws.amazon.com/cli/latest/reference/iot/register-ca-certificate.html) in the *AWS CLI Command Reference*.

```
aws iot register-ca-certificate \
```

```
    --ca-certificate file://root_CA_cert_filename.pem \
    --verification-cert file://verification_cert_filename.pem
```

This command returns the *certificateId*, if successful.

6. At this point, the CA certificate has been registered with AWS IoT but is not active. The CA certificate must be active before you can register any client certificates it has signed.

   This step activates the CA certificate.

   To activate the CA certificate, use the **update-certificate** command as follows. For more information, see update-certificate in the *AWS CLI Command Reference*.

```
aws iot update-ca-certificate \
    --certificate-id certificateId \
    --new-status ACTIVE
```

To see the status of the CA certificate, use the **describe-ca-certificate** command. For more information, see describe-ca-certificate in the *AWS CLI Command Reference*.

Create a CA verification certificate to register the CA certificate in the console

> **Note**
>
> This procedure is only for use if you are registering a CA certificate from the AWS IoT console.
> If you did not come to this procedure from the AWS IoT console, start the CA certificate registration process in the console at Register CA certificate.

Make sure you have the following available on the same computer before you continue:

- The root CA's certificate file (referenced in the following example as *root_CA_cert_filename.pem*)
- The root CA certificate's private key file (referenced in the following example as *root_CA_key_filename.key*)
- OpenSSL v1.1.1i or later

To use the command line interface to create a CA verification certificate to register your CA certificate in the console

1. Replace *verification_cert_key_filename.key* with the name of the verification certificate key file that you want to create (for example, **verification_cert.key**). Then run this command to generate a key pair for the private key verification certificate:

```
openssl genrsa -out verification_cert_key_filename.key 2048
```

2. Replace *verification_cert_key_filename.key* with the name of the key file that you created in step 1.

   Replace *verification_cert_csr_filename.csr* with the name of the certificate signing request (CSR) file that you want to create. For example, **verification_cert.csr**.

   Run this command to create the CSR file.

```
openssl req -new \
    -key verification_cert_key_filename.key \
    -out verification_cert_csr_filename.csr
```

   The command prompts you for additional information that's explained later.

3. In the AWS IoT console, in the **Verification certificate** container, copy the registration code.

4. The information that the **openssl** command prompts you for is shown in the following example. Except for the `Common Name` field, you can enter your own values or keep them blank.

   In the `Common Name` field, paste the registration code that you copied in the previous step.

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
    State or Province Name (full name) []:
    Locality Name (for example, city) []:
    Organization Name (for example, company) []:
```

```
    Organizational Unit Name (for example, section) []:
    Common Name (e.g. server FQDN or YOUR name) []:your_registration_code
    Email Address []:

    Please enter the following 'extra' attributes
    to be sent with your certificate request
    A challenge password []:
    An optional company name []:
```

After you finish, the command creates the CSR file.

5. Replace *verification_cert_csr_filename.csr* with the *verification_cert_csr_filename.csr* you used in the previous step.

   Replace *root_CA_cert_filename.pem* with the file name of the CA certificate that you want to register.

   Replace *root_CA_key_filename.key* with the file name of the CA certificate's private key file.

   Replace *verification_cert_filename.pem* with the file name of the verification certificate that you want to create. For example, **verification_cert.pem**.

```
openssl x509 -req \
    -in verification_cert_csr_filename.csr \
    -CA root_CA_cert_filename.pem \
    -CAkey root_CA_key_filename.key \
    -CAcreateserial \
    -out verification_cert_filename.pem \
    -days 500 -sha256
```

6. After the OpenSSL command completes, you should have these files ready to use for when you return to the console.

   - Your CA certificate file (*root_CA_cert_filename.pem* used in the previous command)
   - The verification certificate that you created in the previous step (*verification_cert_filename.pem* used in the previous command)

Deactivate a CA certificate

When a certificate authority (CA) certificate is enabled for automatic client certificate registration, AWS IoT checks the CA certificate to make sure the CA is `ACTIVE`. If the CA certificate is `INACTIVE`, AWS IoT doesn't allow the client certificate to be registered.

By setting the CA certificate to `INACTIVE`, you prevent any new client certificates issued by the CA from being registered automatically.

> **Note**
>
> Any registered client certificates that were signed by the compromised CA certificate continue to work until you explicitly revoke each one of them.

Deactivate a CA certificate (console)

To deactivate a CA certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **CAs**.
3. In the list of certificate authorities, find the one that you want to deactivate, and choose the ellipsis icon to open the option menu.
4. On the option menu, choose **Deactivate**.

The certificate authority should show as **Inactive** in the list.

> **Note**
>
> The AWS IoT console does not provide a way to list the certificates that were signed by the CA you deactivated. For an AWS CLI option to list those certificates, see Deactivate a CA certificate (CLI).

Deactivate a CA certificate (CLI)

The AWS CLI provides the **update-ca-certificate** command to deactivate a CA certificate.

```
aws iot update-ca-certificate \
```

```
    --certificate-id certificateId \
    --new-status INACTIVE
```

Use the **list-certificates-by-ca** command to get a list of all registered client certificates that were signed by the specified CA. For each client certificate signed by the specified CA certificate, use the **update-certificate** command to revoke the client certificate to prevent it from being used.

Use the **describe-ca-certificate** command to see the status of the CA certificate.

Create a client certificate using your CA certificate

You can use your own certificate authority (CA) to create client certificates. The client certificate must be registered with AWS IoT before use. For information about the registration options for your client certificates, see Register a client certificate.

Create a client certificate (CLI)

> **Note**
>
> You can't perform this procedure in the AWS IoT console.

To create a client certificate using the AWS CLI

1. Generate a key pair.

```
openssl genrsa -out device_cert_key_filename.key 2048
```

2. Create a CSR for the client certificate.

```
openssl req -new \
    -key device_cert_key_filename.key \
    -out device_cert_csr_filename.csr
```

You are prompted for some information, as shown here:

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
```

```
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
    State or Province Name (full name) []:
    Locality Name (for example, city) []:
    Organization Name (for example, company) []:
    Organizational Unit Name (for example, section) []:
    Common Name (e.g. server FQDN or YOUR name) []:
    Email Address []:

    Please enter the following 'extra' attributes
    to be sent with your certificate request
    A challenge password []:
    An optional company name []:
```

3. Create a client certificate from the CSR.

```
openssl x509 -req \
    -in device_cert_csr_filename.csr \
    -CA root_CA_cert_filename.pem \
    -CAkey root_CA_key_filename.key \
    -CAcreateserial \
    -out device_cert_filename.pem \
    -days 500 -sha256
```

At this point, the client certificate has been created, but it has not yet been registered with AWS IoT. For information about how and when to register the client certificate, see Register a client certificate.

Register a client certificate

Client certificates must be registered with AWS IoT to enable communications between the client and AWS IoT. You can register each client certificate manually, or you can configure the client certificates to register automatically when the client connects to AWS IoT for the first time.

If you want your clients and devices to register their client certificates when they first connect, you must Register your CA certificate used to sign the client certificate with AWS IoT in the Regions in which you want to use it. The Amazon Root CA is automatically registered with AWS IoT.

Client certificates can be shared by AWS accounts and Regions. The procedures in these topics must be performed in each account and Region in which you want to use the client certificate.

The registration of a client certificate in one account or Region is not automatically recognized by another.

> **Note**
>
> Clients that use the Transport Layer Security (TLS) protocol to connect to AWS IoT must support the Server Name Indication (SNI) extension to TLS. For more information, see Transport security in AWS IoT Core.

Topics

Register a client certificate manually

You can register a client certificate manually by using the AWS IoT console and AWS CLI.

The registration procedure to use depends on whether the certificate will be shared by AWS accounts and Regions. The registration of a client certificate in one account or Region is not automatically recognized by another.

The procedures in this topic must be performed in each account and Region in which you want to use the client certificate. Client certificates can be shared by AWS accounts and Regions.

Register a client certificate signed by a registered CA (console)

> **Note**
>
> Before you perform this procedure, make sure that you have the client certificate's .pem file and that the client certificate was signed by a CA that you have registered with AWS IoT.

To register an existing certificate with AWS IoT using the console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the navigation pane, under the **Manage** section, choose **Security**, and then choose **Certificates**.

3. On the **Certificates** page in the **Certificates** dialog box, choose **Add certificate**, and then choose **Register certificates**.
4. On the **Register certificate** page in the **Certificates to upload** dialog box, do the following:
   - Choose **CA is registered with AWS IoT**.
   - From **Choose a CA certificate**, select your **Certification authority**.
     - Choose **Register a new CA** to register a new **Certification authority** that's not registered with AWS IoT.
     - Leave **Choose a CA certificate** blank if **Amazon Root certificate authority** is your certification authority.
   - Select up to 10 certificates to upload and register with AWS IoT.
     - Use the certificate files you created in Create AWS IoT client certificates and Create a client certificate using your CA certificate.
   - Choose **Activate** or **Deactivate**. If you choose **Deactive**, Activate or deactivate a client certificate explains how to activate your certificate after certificate registration.
   - Choose **Register**.

On the **Certificates** page in the **Certificates** dialog box, your registered certificates will now appear.

Register a client certificate signed by an unregistered CA (console)

> **Note**
>
> Before you perform this procedure, make sure that you have the client certificate's .pem file.

To register an existing certificate with AWS IoT using the console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**, and then choose **Create**.
3. On **Create a certificate**, locate the **Use my certificate** entry, and choose **Get started**.
4. On **Select a CA**, choose **Next**.
5. On **Register existing device certificates**, choose **Select certificates**, and select up to 10 certificate files to register.

6. After closing the file dialog box, select whether you want to activate or revoke the client certificates when you register them.

   If you don't activate a certificate when it is registered, Activate a client certificate (console) describes how to activate it later.

   If a certificate is revoked when it is registered, it can't be activated later.

   After you choose the certificate files to register, and select the actions to take after registration, select **Register certificates**.

The client certificates that are registered successfully appear in the list of certificates.

Register a client certificate signed by a registered CA (CLI)

> **Note**
>
> Before you perform this procedure, make sure that you have the certificate authority (CA) .pem and the client certificate's .pem file. The client certificate must be signed by a certificate authority (CA) that you have registered with AWS IoT.

Use the **register-certificate** command to register, but not activate, a client certificate.

```
aws iot register-certificate \
    --certificate-pem file://device_cert_filename.pem \
    --ca-certificate-pem file://ca_cert_filename.pem
```

The client certificate is registered with AWS IoT, but it is not active yet. See Activate a client certificate (CLI) for information on how to activate it later.

You can also activate the client certificate when you register it by using this command.

```
aws iot register-certificate \
    --set-as-active \
    --certificate-pem file://device_cert_filename.pem \
    --ca-certificate-pem file://ca_cert_filename.pem
```

For more information about activating the certificate so that it can be used to connect to AWS IoT, see Activate or deactivate a client certificate

Register a client certificate signed by an unregistered CA (CLI)

> **Note**
>
> Before you perform this procedure, make sure that you have the certificate's .pem file.

Use the **register-certificate-without-ca** command to register, but not activate, a client certificate.

```
aws iot register-certificate-without-ca \
    --certificate-pem file://device_cert_filename.pem
```

The client certificate is registered with AWS IoT, but it is not active yet. See Activate a client certificate (CLI) for information on how to activate it later.

You can also activate the client certificate when you register it by using this command.

```
aws iot register-certificate-without-ca \
    --status ACTIVE \
    --certificate-pem file://device_cert_filename.pem
```

For more information about activating the certificate so that it can be used to connect to AWS IoT, see Activate or deactivate a client certificate.

Register a client certificate when the client connects to AWS IoT just-in-time registration (JITR)

You can configure a CA certificate to enable client certificates it has signed to register with AWS IoT automatically the first time the client connects to AWS IoT.

To register client certificates when a client connects to AWS IoT for the first time, you must enable the CA certificate for automatic registration and configure the first connection by the client to provide the required certificates.

Configure a CA certificate to support automatic registration (console)

To configure a CA certificate to support automatic client certificate registration using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **CAs**.

3. In the list of certificate authorities, find the one for which you want to enable automatic registration, and open the option menu by using the ellipsis icon.
4. On the option menu, choose **Enable auto-registration**.

> **Note**
>
> The auto-registration status is not shown in the list of certificate authorities. To see the auto-registration status of a certificate authority, you must open the **Details** page of the certificate authority.

Configure a CA certificate to support automatic registration (CLI)

If you have already registered your CA certificate with AWS IoT, use the **update-ca-certificate** command to set `autoRegistrationStatus` of the CA certificate to `ENABLE`.

```
aws iot update-ca-certificate \
--certificate-id caCertificateId \
--new-auto-registration-status ENABLE
```

If you want to enable `autoRegistrationStatus` when you register the CA certificate, use the **register-ca-certificate** command.

```
aws iot register-ca-certificate \
--allow-auto-registration  \
--ca-certificate file://root_CA_cert_filename.pem \
--verification-cert file://verification_cert_filename.pem
```

Use the **describe-ca-certificate** command to see the status of the CA certificate.

Configure the first connection by a client for automatic registration

When a client attempts to connect to AWS IoT for the first time, the client certificate signed by your CA certificate must be present on the client during the Transport Layer Security (TLS) handshake.

When the client connects to AWS IoT, use the client certificate you created in Create AWS IoT client certificates or Create your own client certificates. AWS IoT recognizes the CA certificate as a registered CA certificate, registers the client certificate, and sets its status to

PENDING_ACTIVATION. This means that the client certificate was automatically registered and is awaiting activation. The client certificate's state must be ACTIVE before it can be used to connect to AWS IoT. See Activate or deactivate a client certificate for information on activating a client certificate.

> **Note**
>
> You can provision devices using AWS IoT Core just-in-time registration (JITR) feature without having to send the entire trust chain on devices' first connection to AWS IoT Core. Presenting the CA certificate is optional but the device is required to send the Server Name Indication (SNI) extension when they connect.

When AWS IoT automatically registers a certificate or when a client presents a certificate in the PENDING_ACTIVATION status, AWS IoT publishes a message to the following MQTT topic:

`$aws/events/certificates/registered/caCertificateId`

Where *caCertificateId* is the ID of the CA certificate that issued the client certificate.

The message published to this topic has the following structure:

```
{
        "certificateId": "certificateId",
        "caCertificateId": "caCertificateId",
        "timestamp": timestamp,
        "certificateStatus": "PENDING_ACTIVATION",
        "awsAccountId": "awsAccountId",
        "certificateRegistrationTimestamp": "certificateRegistrationTimestamp"
}
```

You can create a rule that listens on this topic and performs some actions. We recommend that you create a Lambda rule that verifies the client certificate is not on a certificate revocation list (CRL), activates the certificate, and creates and attaches a policy to the certificate. The policy determines which resources the client can access. If the policy you are creating requires the client ID from the connecting devices, you can use rule's clientid() function to retrieve the client ID. An example rule definition can look like the following:

```
SELECT *,
   clientid() as clientid
```

```
from $aws/events/certificates/registered/caCertificateId
```

In this example, the rule subscribes to the JITR topic `$aws/events/certificates/registered/caCertificateID` and uses the clientid() function to retrieve the client ID. The rule then appends the client ID to the JITR payload. For more information about rule's clientid() function, see clientid().

For more information about how to create a Lambda rule that listens on the `$aws/events/certificates/registered/caCertificateID` topic and performs these actions, see just-in-time registration of Client Certificates on AWS IoT.

If any error or exception occurs during the auto-registration of the client certificates, AWS IoT sends events or messages to your logs in CloudWatch Logs. For more information about setting up the logs for your account, see the Amazon CloudWatch documentation.

Manage client certificates

AWS IoT provides capabilities for you to manage client certificates.

In this topic:

- Activate or deactivate a client certificate
- Attach a thing or policy to a client certificate
- Revoke a client certificate
- Transfer a certificate to another account

Activate or deactivate a client certificate

AWS IoT verifies that a client certificate is active when it authenticates a connection.

You can create and register client certificates without activating them so they can't be used until you want to use them. You can also deactivate active client certificates to disable them temporarily. Finally, you can revoke client certificates to prevent them from any future use.

Activate a client certificate (console)

To activate a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

3. In the list of certificates, locate the certificate that you want to activate, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Activate**.

The certificate should show as **Active** in the list of certificates.

Deactivate a client certificate (console)

To deactivate a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the [AWS IoT console](#).
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate that you want to deactivate, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Deactivate**.

The certificate should show as **Inactive** in the list of certificates.

Activate a client certificate (CLI)

The AWS CLI provides the **update-certificate** command to activate a certificate.

```
aws iot update-certificate \
    --certificate-id certificateId \
    --new-status ACTIVE
```

If the command was successful, the certificate's status will be `ACTIVE`. Run **describe-certificate** to see the certificate's status.

```
aws iot describe-certificate \
    --certificate-id certificateId
```

Deactivate a client certificate (CLI)

The AWS CLI provides the **update-certificate** command to deactivate a certificate.

```
aws iot update-certificate \
    --certificate-id certificateId \
```

```
    --new-status INACTIVE
```

If the command was successful, the certificate's status will be INACTIVE. Run **describe-certificate** to see the certificate's status.

```
aws iot describe-certificate \
    --certificate-id certificateId
```

Attach a thing or policy to a client certificate

When you create and register a certificate separate from an AWS IoT thing, it will not have any policies that authorize any AWS IoT operations, nor will it be associated with any AWS IoT thing object. This section describes how to add these relationships to a registered certificate.

> **Important**
>
> To complete these procedures, you must have already created the thing or policy that you want to attach to the certificate.

The certificate authenticates a device with AWS IoT so that it can connect. Attaching the certificate to a thing resource establishes the relationship between the device (by way of the certificate) and the thing resource. To authorize the device to perform AWS IoT actions, such as to allow the device to connect and publish messages, an appropriate policy must be attached to the device's certificate.

Attach a thing to a client certificate (console)

You will need the name of the thing object to complete this procedure.

To attach a thing object to a registered certificate

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate to which you want to attach a policy, open the certificate's option menu by choosing the ellipsis icon, and choose **Attach thing**.
4. In the pop-up, locate the name of the thing you want to attach to the certificate, choose its check box, and choose **Attach**.

The thing object should now appear in the list of things on the certificate's details page.

Attach a policy to a client certificate (console)

You will need the name of the policy object to complete this procedure.

To attach a policy object to a registered certificate

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate to which you want to attach a policy, open the certificate's option menu by choosing the ellipsis icon, and choose **Attach policy**.
4. In the pop-up, locate the name of the policy you want to attach to the certificate, choose its check box, and choose **Attach**.

The policy object should now appear in the list of policies on the certificate's details page.

Attach a thing to a client certificate (CLI)

The AWS CLI provides the **attach-thing-principal** command to attach a thing object to a certificate.

```
aws iot attach-thing-principal \
    --principal certificateArn \
    --thing-name thingName
```

Attach a policy to a client certificate (CLI)

The AWS CLI provides the **attach-policy** command to attach a policy object to a certificate.

```
aws iot attach-policy \
    --target certificateArn \
    --policy-name policyName
```

Revoke a client certificate

If you detect suspicious activity on a registered client certificate, you can revoke it so that it can't be used again.

> **Note**
>
> Once a certificate is revoked, it's status can't be changed. That is, the certificate status can't be changed to `Active` or any other status.

Revoke a client certificate (console)

To revoke a client certificate using the AWS IoT console

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.
3. In the list of certificates, locate the certificate that you want to revoke, and open the option menu by using the ellipsis icon.
4. In the option menu, choose **Revoke**.

If the certificate was successfully revoked, it will show as **Revoked** in the list of certificates.

Revoke a client certificate (CLI)

The AWS CLI provides the **update-certificate** command to revoke a certificate.

```
aws iot update-certificate \
    --certificate-id certificateId \
    --new-status REVOKED
```

If the command was successful, the certificate's status will be `REVOKED`. Run **describe-certificate** to see the certificate's status.

```
aws iot describe-certificate \
    --certificate-id certificateId
```

Transfer a certificate to another account

X.509 certificates that belong to one AWS account can be transferred to another AWS account.

To transfer an X.509 certificate from one AWS account to another

1. the section called "Begin a certificate transfer"

The certificate must be deactivated and detached from all policies and things before initiating the transfer.

2. [the section called “Accept or reject a certificate transfer”]

   The receiving account must explicitly accept or reject the transferred certificate. After the receiving account accepts the certificate, the certificate must be activated before use.

3. [the section called “Cancel a certificate transfer”]

   The originating account can cancel a transfer, if the certificate has not been accepted.

Begin a certificate transfer

You can begin to transfer a certificate to another AWS account by using the AWS IoT console or the AWS CLI.

Begin a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate that you want to transfer.

Do this procedure from the account with the certificate to transfer.

To begin to transfer a certificate to another AWS account

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

   Choose the certificate with an **Active** or **Inactive** status that you want to transfer and open its details page.
3. On the certificate's **Details** page, in the **Actions** menu, if the **Deactivate** option is available, choose the **Deactivate** option to deactivate the certificate.
4. On the certificate's **Details** page, in the left menu, choose **Policies**.
5. On the certificate's **Policies** page, if there are any policies attached to the certificate, detach each one by opening the policy's options menu and choosing **Detach**.

   The certificate must not have any attached policies before you continue.
6. On the certificate's **Policies** page, in the left menu, choose **Things**.
7. On the certificate's **Things** page, if there are any things attached to the certificate, detach each one by opening the thing's options menu and choosing **Detach**.

The certificate must not have any attached things before you continue.

8. On the certificate's **Things** page, in the left menu, choose **Details**.
9. On the certificate's **Details** page, in the **Actions** menu, choose **Start transfer** to open the **Start transfer** dialog box.
10. In the **Start transfer** dialog box, enter the AWS account number of the account to receive the certificate and an optional short message.
11. Choose **Start transfer** to transfer the certificate.

The console should display a message that indicates the success or failure of the transfer. If the transfer was started, the certificate's status is updated to **Transferred**.

Begin a certificate transfer (CLI)

To complete this procedure, you'll need the *certificateId* and the *certificateArn* of the certificate that you want to transfer.

Do this procedure from the account with the certificate to transfer.

To begin to transfer a certificate to another AWS account

1. Use the update-certificate command to deactivate the certificate.

```
aws iot update-certificate --certificate-id certificateId --new-status INACTIVE
```

2. Detach all policies.

   1. Use the list-attached-policies command to list the policies attached to the certificate.

   ```
   aws iot list-attached-policies --target certificateArn
   ```

   2. For each attached policy, use the detach-policy command to detach the policy.

   ```
   aws iot detach-policy --target certificateArn --policy-name policy-name
   ```

3. Detach all things.

   1. Use the list-principal-things command to list the things attached to the certificate.

   ```
   aws iot list-principal-things --principal certificateArn
   ```

2. For each attached thing, use the **detach-thing-principal** command to detach the thing.

```
aws iot detach-thing-principal --principal certificateArn --thing-name thing-name
```

4. Use the **transfer-certificate** command to start the certificate transfer.

```
aws iot transfer-certificate --certificate-id certificateId --target-aws-account account-id
```

Accept or reject a certificate transfer

You can accept or reject a certificate transferred to you AWS account from another AWS account by using the AWS IoT console or the AWS CLI.

Accept or reject a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate that was transferred to your account.

Do this procedure from the account receiving the certificate that was transferred.

To accept or reject a certificate that was transferred to your AWS account

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

   Choose the certificate with a status of **Pending transfer** that you want to accept or reject and open its details page.
3. On the certificate's **Details** page, in the **Actions** menu,
   - To accept the certificate, choose **Accept transfer**.
   - To not accept the certificate, choose **Reject transfer**.

Accept or reject a certificate transfer (CLI)

To complete this procedure, you'll need the `certificateId` of the certificate transfer that you want to accept or reject.

Do this procedure from the account receiving the certificate that was transferred.

To accept or reject a certificate that was transferred to your AWS account

1. Use the accept-certificate-transfer command to accept the certificate.

```
aws iot accept-certificate-transfer --certificate-id certificateId
```

2. Use the reject-certificate-transfer command to reject the certificate.

```
aws iot reject-certificate-transfer --certificate-id certificateId
```

Cancel a certificate transfer

You can cancel a certificate transfer before it has been accepted by using the AWS IoT console or the AWS CLI.

Cancel a certificate transfer (console)

To complete this procedure, you'll need the ID of the certificate transfer that you want to cancel.

Do this procedure from the account that initiated the certificate transfer.

To cancel a certificate transfer

1. Sign in to the AWS Management Console and open the AWS IoT console.
2. In the left navigation pane, choose **Secure**, choose **Certificates**.

   Choose the certificate with **Transferred** status whose transfer you want to cancel and open its options menu.
3. On the certificate's options menu, choose the **Revoke transfer** option to cancel the certificate transfer.

> **Important**
>
> Be careful not to mistake the **Revoke transfer** option with the **Revoke** option. The **Revoke transfer** option cancels the certificate transfer, while the **Revoke** option makes the certificate irreversibly unusable by AWS IoT.

Cancel a certificate transfer (CLI)

To complete this procedure, you'll need the *certificateId* of the certificate transfer that you want to cancel.

Do this procedure from the account that initiated the certificate transfer.

Use the **cancel-certificate-transfer** command to cancel the certificate transfer.

```
aws iot cancel-certificate-transfer --certificate-id certificateId
```

Custom client certificate validation

AWS IoT Core supports custom client certificate validation for X.509 client certificates, which enhances client authentication management. This certificate validation method is also known as pre-authentication certificate checks, in which you evaluate client certificates based on your own criteria (defined in a Lambda function) and revoke client certificates or the certificates' signing certificate authority (CA) certificate to prevent clients to connect to AWS IoT Core. For example, you can create your own certificate revocation checks that validate the certificates' status against validation authorities that support Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRL) endpoints, and prevent connections for clients with revoked certificates. The criteria used to evaluate client certificates are defined in a Lambda function (also known as pre-authentication Lambda). You must use the endpoints set in domain configurations and the authentication type must be X.509 certificate. In addition, clients must provide the Server Name Indication (SNI) extension when connecting to AWS IoT Core.

> **Note**
>
> This feature is not supported in the AWS GovCloud (US) Regions.

The process of performing custom client certificate validation involves the following steps.

- Step 1: Register your X.509 client certificates with AWS IoT Core
- Step 2: Create a Lambda function
- Step 3: Authorize AWS IoT to invoke your Lambda function
- Step 4: Set authentication configuration for a domain

Step 1: Register your X.509 client certificates with AWS IoT Core

If you haven't done this already, register and activate your X.509 client certificates with AWS IoT Core. Otherwise, skip to the next step.

To register and activate your client certificates with AWS IoT Core, follow the steps:

1. If you create client certificates directly with AWS IoT. These client certificates will be automatically registered with AWS IoT Core.
2. If you create your own client certificates, follow these instructions to register them with AWS IoT Core.
3. To activate your client certificates, follow these instructions.

Step 2: Create a Lambda function

You need to create a Lambda function that will perform certificate verification and be called for every client connect attempt for the configured endpoint. When creating this Lambda function, follow the general guidance from Create your first Lambda function. Additionally, ensure that the Lambda function adheres to the expected request and response formats as follows:

Lambda function event example

```
{
 "connectionMetadata": {
  "id": "string"
 },
 "principalId": "string",
 "serverName": "string",
 "clientCertificateChain": [
  "string",
  "string"
 ]
}
```

`connectionMetadata`

Metadata or additional information related to the client's connection to AWS IoT Core.

`principalId`

The principal identifier associated with the client in the TLS connection.

`serverName`

The Server Name Indication (SNI) hostname string. AWS IoT Core requires devices to send the SNI extension to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field.

`clientCertificateChain`

The array of strings that represents the client's X.509 certificate chain.

Lambda function response example

```
{
 "isAuthenticated": "boolean"
}
```

`isAuthenticated`

A Boolean value that indicates whether the request is authenticated.

> **Note**
>
> In the Lambda response, `isAuthenticated` must be `true` to proceed to further authentication and authorization. Otherwise, the IoT client certificate can be disabled and custom authentication with X.509 client certificates can be blocked for further authentication and authorization.

Step 3: Authorize AWS IoT to invoke your Lambda function

After creating the Lambda function, you must grant permission for AWS IoT to invoke it, by using the add-permission CLI command. Note that this Lambda function will be invoked for every connect attempt to your configured endpoint. For more information, see Authorizing AWS IoT to invoke your Lambda function.

Step 4: Set authentication configuration for a domain

The following section describes how to set authentication configuration for a custom domain using the AWS CLI.

Set client certificate configuration for a domain (CLI)

If you don't have a domain configuration, use the **create-domain-configuration** CLI command to create one. If you already have a domain configuration, use the **update-domain-configuration** CLI command to update the client certificate configuration for a domain. You must add the ARN of the Lambda function that you've created in the previous step.

```
aws iot create-domain-configuration \
    --domain-configuration-name domainConfigurationName \
    --authentication-type AWS_X509|CUSTOM_AUTH_X509 \
    --application-protocol SECURE_MQTT|HTTPS \
    --client-certificate-config 'clientCertificateCallbackArn":"arn:aws:lambda:us-
east-2:123456789012:function:my-function:1"}'
```

```
aws iot update-domain-configuration \
    --domain-configuration-name domainConfigurationName \
    --authentication-type AWS_X509|CUSTOM_AUTH_X509 \
    --application-protocol SECURE_MQTT|HTTPS \
    --client-certificate-config '{"clientCertificateCallbackArn":"arn:aws:lambda:us-
east-2:123456789012:function:my-function:1"}'
```

`domain-configuration-name`

The name of the domain configuration.

`authentication-type`

The authentication type of the domain configuration. For more information, see choosing an authentication type.

`application-protocol`

The application protocol which devices use to communicate with AWS IoT Core. For more information, see choosing an application protocol.

`client-certificate-config`

An object that specifies the client authentication configuration for a domain.

`clientCertificateCallbackArn`

The Amazon Resource Name (ARN) of the Lambda function that AWS IoT invokes in TLS layer when new connection is being established. To customize client authentication to perform custom client certificate validation, you must add the ARN of the Lambda function that you've created in the previous step.

For more information, see CreateDomainConfiguration and UpdateDomainConfiguration from the *AWS IoT API Reference*. For more information about domain configurations, see Domain configurations.

IAM users, groups, and roles

IAM users, groups, and roles are the standard mechanisms for managing identity and authentication in AWS. You can use them to connect to AWS IoT HTTP interfaces using the AWS SDK and AWS CLI.

IAM roles also allow AWS IoT to access other AWS resources in your account on your behalf. For example, if you want to have a device publish its state to a DynamoDB table, IAM roles allow AWS IoT to interact with Amazon DynamoDB. For more information, see IAM Roles.

For message broker connections over HTTP, AWS IoT authenticates users, groups, and roles using the Signature Version 4 signing process. For information, see Signing AWS API Requests.

When using AWS Signature Version 4 with AWS IoT, clients must support the following in their TLS implementation:

- TLS 1.2
- SHA-256 RSA certificate signature validation
- One of the cipher suites from the TLS cipher suite support section

For information, see Identity and access management for AWS IoT.

Amazon Cognito identities

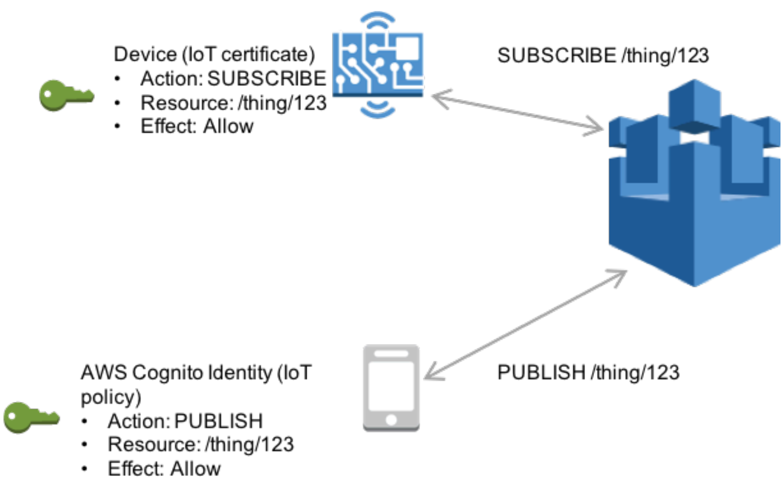
Amazon Cognito Identity enables you to create temporary, limited privilege AWS credentials for use in mobile and web applications. When you use Amazon Cognito Identity, create identity pools that create unique identities for your users and authenticate them with identity providers like Login with Amazon, Facebook, and Google. You can also use Amazon Cognito identities with your own developer authenticated identities. For more information, see Amazon Cognito Identity.

To use Amazon Cognito Identity, define an Amazon Cognito identity pool that is associated with an IAM role. The IAM role is associated with an IAM policy that grants identities from your identity pool permission to access AWS resources like calling AWS services.

Amazon Cognito Identity creates unauthenticated and authenticated identities. Unauthenticated identities are used for guest users in a mobile or web application who want to use the app without signing in. Unauthenticated users are granted only those permissions specified in the IAM policy associated with the identity pool.

When you use authenticated identities, in addition to the IAM policy attached to the identity pool, you must attach an AWS IoT policy to an Amazon Cognito Identity. To attach an AWS IoT policy, use the AttachPolicy API and give permissions to an individual user of your AWS IoT application. You can use the AWS IoT policy to assign fine-grained permissions for specific customers and their devices.

Authenticated and unauthenticated users are different identity types. If you don't attach an AWS IoT policy to the Amazon Cognito Identity, an authenticated user fails authorization in AWS IoT and doesn't have access to AWS IoT resources and actions. For more information about creating policies for Amazon Cognito identities, see Publish/Subscribe policy examples and Authorization with Amazon Cognito identities.



Custom authentication and authorization

AWS IoT Core lets you define custom authorizers so that you can manage your own client authentication and authorization. This is useful when you need to use authentication mechanisms other than the ones that AWS IoT Core natively supports. (For more information about the natively supported mechanisms, see the section called “Client authentication”).

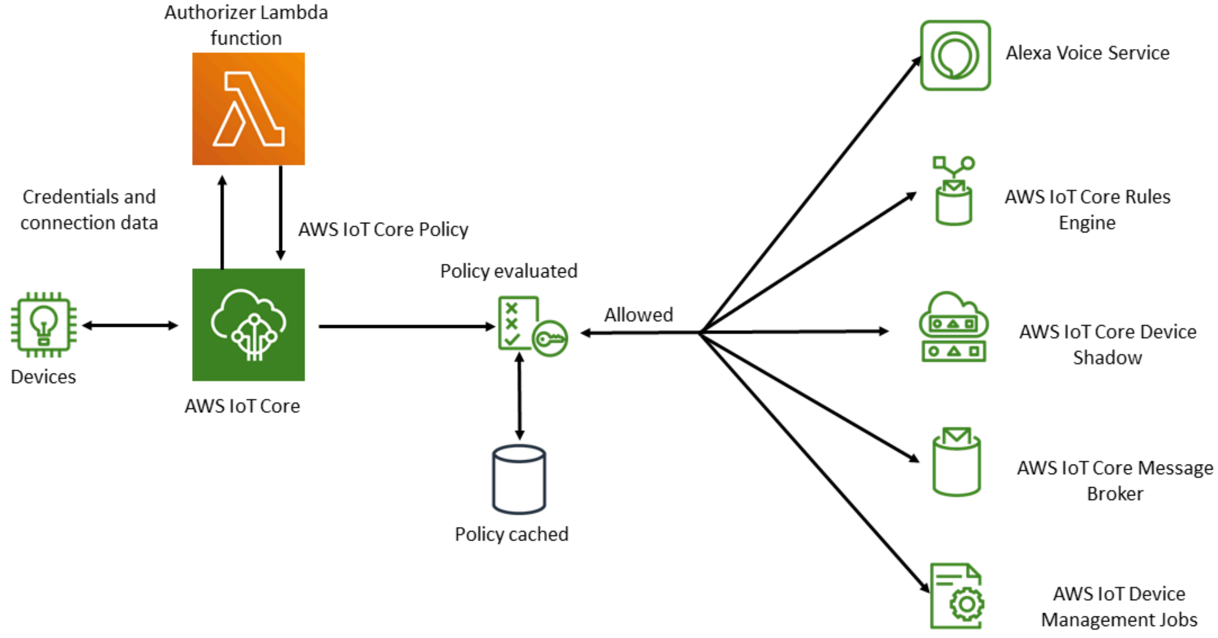
For example, if you are migrating existing devices in the field to AWS IoT Core and these devices use a custom bearer token or MQTT user name and password to authenticate, you can migrate them to AWS IoT Core without having to provision new identities for them. You can use custom authentication with any of the communication protocols that AWS IoT Core supports. For more information about the protocols that AWS IoT Core supports, see the section called "Device communication protocols".

Topics

- Understanding the custom authentication workflow
- Creating and managing custom authorizers (CLI)
- Custom authentication with X.509 client certificates
- Connecting to AWS IoT Core by using custom authentication
- Troubleshooting your authorizers

Understanding the custom authentication workflow

Custom authentication enables you to define how to authenticate and authorize clients by using authorizer resources. Each authorizer contains a reference to a customer-managed Lambda function, an optional public key for validating device credentials, and additional configuration information. The following diagram illustrates the authorization workflow for custom authentication in AWS IoT Core.



AWS IoT Core custom authentication and authorization workflow

The following list explains each step in the custom authentication and authorization workflow.

1. A device connects to a customer's AWS IoT Core data endpoint by using one of the supported the section called "Device communication protocols". The device passes credentials in either the request's header fields or query parameters (for the HTTP Publish or MQTT over WebSockets protocols), or in the user name and password field of the MQTT CONNECT message (for the MQTT and MQTT over WebSockets protocols).
2. AWS IoT Core checks for one of two conditions:
   - The incoming request specifies an authorizer.
   - The AWS IoT Core data endpoint receiving the request has a default authorizer configured for it.

   If AWS IoT Core finds an authorizer in either of these ways, AWS IoT Core triggers the Lambda function associated with the authorizer.
3. (Optional) If you've enabled token signing, AWS IoT Core validates the request signature by using the public key stored in the authorizer before triggering the Lambda function. If validation fails, AWS IoT Core stops the request without invoking the Lambda function.

4. The Lambda function receives the credentials and connection metadata in the request and makes an authentication decision.
5. The Lambda function returns the results of the authentication decision and an AWS IoT Core policy document that specifies what actions are allowed in the connection. The Lambda function also returns information that specifies how often AWS IoT Core revalidates the credentials in the request by invoking the Lambda function.
6. AWS IoT Core evaluates activity on the connection against the policy it has received from the Lambda function.
7. After the connection is established and your custom authorizer Lambda is initially invoked, the next invocation can be delayed for up to 5 minutes on idle connections without any MQTT operations. After that, subsequent invocations will follow the refresh interval in your custom authorizer Lambda. This approach can prevent excessive invocations that could exceed the Lambda concurrency limit of your AWS account.

Scaling considerations

Because a Lambda function handles authentication and authorization for your authorizer, the function is subject to Lambda pricing and service limits, such as concurrent execution rate. For more information about Lambda pricing, see Lambda Pricing. You can manage the load on your Lambda function by adjusting the `refreshAfterInSeconds` and `disconnectAfterInSeconds` parameters in your Lambda function response. For more information about the contents of your Lambda function response, see the section called “Defining your Lambda function”.

> **Note**
>
> If you leave signing enabled, you can prevent excessive triggering of your Lambda by unrecognized clients. Consider this before you disable signing in your authorizer.

> **Note**
>
> The Lambda function timeout limit for custom authorizer is 5 seconds.

Creating and managing custom authorizers (CLI)

AWS IoT Core implements custom authentication and authorization schemes by using custom authorizers. A custom authorizer is an AWS IoT Core resource that gives you the flexibility to define and implement the rules and policies based on your specific requirements. To create a custom authorizer with step-by-step instructions, see Tutorial: Creating a custom authorizer for AWS IoT Core.

Each authorizer consists of the following components:

- *Name*: A unique user-defined string that identifies the authorizer.
- *Lambda function ARN*: The Amazon Resource Name (ARN) of the Lambda function that implements the authorization and authentication logic.
- *Token key name*: The key name used to extract the token from the HTTP headers, query parameters, or MQTT CONNECT user name in order to perform signature validation. This value is required if signing is enabled in your authorizer.
- *Signing disabled flag (optional)*: A Boolean value that specifies whether to disable the signing requirement on credentials. This is useful for scenarios where signing the credentials doesn't make sense, such as authentication schemes that use MQTT user name and password. The default value is `false`, so signing is enabled by default.
- *Token signing public key*: The public key that AWS IoT Core uses to validate the token signature. Its minimum length is 2,048 bits. This value is required if signing is enabled in your authorizer.

Lambda charges you for the number of times your Lambda function runs and for the amount of time it takes for the code in your function to execute. For more information about Lambda pricing, see Lambda Pricing. For more information about creating Lambda functions, see the Lambda Developer Guide.

> **Note**
>
> If you leave signing enabled, you can prevent excessive triggering of your Lambda by unrecognized clients. Consider this before you disable signing in your authorizer.

> **Note**
>
> The Lambda function timeout limit for custom authorizer is 5 seconds.

In this chapter:

Defining your Lambda function

When AWS IoT Core invokes your authorizer, it triggers the associated Lambda associated with the authorizer with an event that contains the following JSON object. The example JSON object contains all of the possible fields. Any fields that aren't relevant to the connection request aren't included.

```
{
    "token" :"aToken",
    "signatureVerified": Boolean, // Indicates whether the device gateway has validated
 the signature.
    "protocols": ["tls", "http", "mqtt"], // Indicates which protocols to expect for
 the request.
    "protocolData": {
        "tls" : {
            "serverName": "serverName" // The server name indication (SNI) host_name
 string.
        },
        "http": {
            "headers": {
                "#{name}": "#{value}"
            },
            "queryString": "?#{name}=#{value}"
        },
        "mqtt": {
            "username": "myUserName",
            "password": "myPassword", // A base64-encoded string.
```

```
            "clientId": "myClientId" // Included in the event only when the device
 sends the value.
        }
    },
    "connectionMetadata": {
        "id": UUID // The connection ID. You can use this for logging.
    },
}
```

The Lambda function should use this information to authenticate the incoming connection and decide what actions are permitted in the connection. The function should send a response that contains the following values.

- `isAuthenticated`: A Boolean value that indicates whether the request is authenticated.
- `principalId`: An alphanumeric string that acts as an identifier for the token sent by the custom authorization request. The value must be an alphanumeric string with at least one, and no more than 128, characters and match this regular expression (regex) pattern: `([a-zA-Z0-9]){1,128}`. Special characters that are not alphanumeric are not allowed for use with the `principalId` in AWS IoT Core. Refer to the documentation for other AWS services if non-alphanumeric special characters are allowed for the `principalId`.
- `policyDocuments`: A list of JSON-formatted AWS IoT Core policy documents For more information about creating AWS IoT Core policies, see the section called "AWS IoT Core policies". The maximum number of policy documents is 10 policy documents. Each policy document can contain a maximum of 2,048 characters.
- `disconnectAfterInSeconds`: An integer that specifies the maximum duration (in seconds) of the connection to the AWS IoT Core gateway. The minimum value is 300 seconds, and the maximum value is 86,400 seconds. The default value is 86,400.

> **Note**
>
> The value of `disconnectAfterInSeconds` (returned by the Lambda function) is set when the connection establishes. This value cannot be modified during subsequent policy refresh Lambda invocations.

- `refreshAfterInSeconds`: An integer that specifies the interval between policy refreshes. When this interval passes, AWS IoT Core invokes the Lambda function to allow for policy refreshes. The minimum value is 300 seconds, and the maximum value is 86,400 seconds.

The following JSON object contains an example of a response that your Lambda function can send.


```
{ "isAuthenticated":true, //A Boolean that determines whether
client can connect. "principalId": "xxxxxxxx",  //A string that
identifies the connection in logs. "disconnectAfterInSeconds": 86400,
  "refreshAfterInSeconds": 300,   "policyDocuments": [       {
 "Version": "2012-10-17",         "Statement": [           {
     "Action": "iot:Publish",              "Effect": "Allow",
       "Resource": "arn:aws:iot:us-east-1:<your_aws_account_id>:topic/
customauthtesting"             }          ]        }     ] }
```


The `policyDocument` value must contain a valid AWS IoT Core policy document. For more information about AWS IoT Core policies, see the section called "AWS IoT Core policies". In MQTT over TLS and MQTT over WebSockets connections, AWS IoT Core caches this policy for the interval specified in the value of the `refreshAfterInSeconds` field. In the case of HTTP connections the Lambda function is called for every authorization request unless your device is using HTTP persistent connections (also called HTTP keep-alive or HTTP connection reuse) you can choose to enable caching when configuring the authorizer. During this interval, AWS IoT Core authorizes actions in an established connection against this cached policy without triggering your Lambda function again. If failures occur during custom authentication, AWS IoT Core terminates the connection. AWS IoT Core also terminates the connection if it has been open for longer than the value specified in the `disconnectAfterInSeconds`parameter.

The following JavaScript contains a sample Node.js Lambda function that looks for a password in the MQTT Connect message with a value of `test` and returns a policy that grants permission to connect to AWS IoT Core with a client named `myClientName` and publish to a topic that contains the same client name. If it doesn't find the expected password, it returns a policy that denies those two actions.

```
// A simple Lambda function for an authorizer. It demonstrates
// how to parse an MQTT password and generate a response.

exports.handler = function(event, context, callback) {
    var uname = event.protocolData.mqtt.username;
    var pwd = event.protocolData.mqtt.password;
    var buff = new Buffer(pwd, 'base64');
    var passwd = buff.toString('ascii');
    switch (passwd) {
```

```
        case 'test':
            callback(null, generateAuthResponse(passwd, 'Allow'));
            break;
        default:
            callback(null, generateAuthResponse(passwd, 'Deny'));
    }
};

// Helper function to generate the authorization response.
var generateAuthResponse = function(token, effect) {
    var authResponse = {};
    authResponse.isAuthenticated = true;
    authResponse.principalId = 'TEST123';

    var policyDocument = {};
    policyDocument.Version = '2012-10-17       ';
    policyDocument.Statement = [];
    var publishStatement = {};
    var connectStatement = {};
    connectStatement.Action = ["iot:Connect"];
    connectStatement.Effect = effect;
    connectStatement.Resource = ["arn:aws:iot:us-east-1:123456789012:client/
myClientName"];
    publishStatement.Action = ["iot:Publish"];
    publishStatement.Effect = effect;
    publishStatement.Resource = ["arn:aws:iot:us-east-1:123456789012:topic/telemetry/
myClientName"];
    policyDocument.Statement[0] = connectStatement;
    policyDocument.Statement[1] = publishStatement;
    authResponse.policyDocuments = [policyDocument];
    authResponse.disconnectAfterInSeconds = 3600;
    authResponse.refreshAfterInSeconds = 300;

    return authResponse;
}
```

The preceding Lambda function returns the following JSON when it receives the expected password of `test` in the MQTT Connect message. The values of the `password` and `principalId` properties will be the values from the MQTT Connect message.

```
{
  "password": "password",
  "isAuthenticated": true,
```

```
  "principalId": "principalId",
  "policyDocuments": [
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Action": "iot:Connect",
          "Effect": "Allow",
          "Resource": "*"
        },
        {
          "Action": "iot:Publish",
          "Effect": "Allow",
          "Resource": "arn:aws:iot:region:accountId:topic/telemetry/${iot:ClientId}"
        },
        {
          "Action": "iot:Subscribe",
          "Effect": "Allow",
          "Resource": "arn:aws:iot:region:accountId:topicfilter/telemetry/
${iot:ClientId}"
        },
        {
          "Action": "iot:Receive",
          "Effect": "Allow",
          "Resource": "arn:aws:iot:region:accountId:topic/telemetry/${iot:ClientId}"
        }
      ]
    }
  ],
  "disconnectAfterInSeconds": 3600,
  "refreshAfterInSeconds": 300
}
```

Creating an authorizer

You can create an authorizer by using the [CreateAuthorizer API](). The following example describes the command.

```
aws iot create-authorizer
--authorizer-name MyAuthorizer
--authorizer-function-arn arn:aws:lambda:us-
west-2:<account_id>:function:MyAuthorizerFunction  //The ARN of the Lambda function.
[--token-key-name MyAuthorizerToken //The key used to extract the token from headers.
```

```
[--token-signing-public-keys FirstKey=
 "-----BEGIN PUBLIC KEY-----
  [...insert your public key here...]
  -----END PUBLIC KEY-----"
[--status ACTIVE]
[--tags <value>]
[--signing-disabled | --no-signing-disabled]
```

You can use the `signing-disabled` parameter to opt out of signature validation for each invocation of your authorizer. We strongly recommend that you do not disable signing unless you have to. Signature validation protects you against excessive invocations of your Lambda function from unknown devices. You can't update the `signing-disabled` status of an authorizer after you create it. To change this behavior, you must create another custom authorizer with a different value for the `signing-disabled` parameter.

Values for the `tokenKeyName` and `tokenSigningPublicKeys` parameters are optional if you have disabled signing. They are required values if signing is enabled.

After you create your Lambda function and the custom authorizer, you must explicitly grant the AWS IoT Core service permission to invoke the function on your behalf. You can do this with the following command.

> **Note**
>
> The default IoT endpoint might not support using custom authorizers with Lambda functions. Instead, you can use domain configurations to define a new endpoint and then specify that endpoint for the custom authorizer.

```
aws lambda add-permission --function-name <lambda_function_name>
--principal iot.amazonaws.com --source-arn <authorizer_arn>
--statement-id Id-123 --action "lambda:InvokeFunction"
```

Authorizing AWS IoT to invoke your Lambda function

In this section, you'll grant the permission of the custom authorizer resource that you just created to run the Lambda function. To grant the permission, you can use the add-permission CLI command.

Grant permission to your Lambda function using the AWS CLI

1. After inserting your values, enter the following command. Note that the `statement-id` value must be unique. Replace *Id-1234* with the exact value you have, otherwise, you might get a `ResourceConflictException` error.

```
aws lambda add-permission  \
--function-name "custom-auth-function" \
--principal "iot.amazonaws.com" \
--action "lambda:InvokeFunction" \
--statement-id "Id-1234" \
--source-arn authorizerArn
```

2. If the command succeeds, it returns a permission statement, such as this example. You can continue to the next section to test the custom authorizer.

```
{
    "Statement": "{\"Sid\":\"Id-1234\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:Region:57EXAMPLE833:function:custom-
auth-function\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
\"arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-function\"}}}"
}
```

If the command doesn't succeed, it returns an error, such as this example. You'll need to review and correct the error before you continue.

```
An error occurred (AccessDeniedException) when calling the AddPermission operation:
 User: arn:aws:iam::57EXAMPLE833:user/EXAMPLE-1 is not authorized to perform:
 lambda:AddPer
mission on resource: arn:aws:lambda:Region:57EXAMPLE833:function:custom-auth-
function
```

Testing your authorizers

You can use the [TestInvokeAuthorizer](https://docs.aws.amazon.com/iot/latest/apireference/API_TestInvokeAuthorizer.html) API to test the invocation and return values of your authorizer. This API enables you to specify protocol metadata and test the signature validation in your authorizer.

The following tabs show how to use the AWS CLI to test your authorizer.

Unix-like

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER \
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

Windows CMD

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER ^
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

Windows PowerShell

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER `
--token TOKEN_VALUE --token-signature TOKEN_SIGNATURE
```

The value of the `token-signature` parameter is the signed token. To learn how to obtain this value, see the section called "Signing the token".

If your authorizer takes a user name and password, you can pass this information by using the `--mqtt-context` parameter. The following tabs show how to use the `TestInvokeAuthorizer` API to send a JSON object that contains a user name, password, and client name to your custom authorizer.

Unix-like

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER  \
--mqtt-context '{"username": "USER_NAME", "password": "dGVzdA==",
 "clientId":"CLIENT_NAME"}'
```

Windows CMD

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER  ^
--mqtt-context '{"username": "USER_NAME", "password": "dGVzdA==",
 "clientId":"CLIENT_NAME"}'
```

Windows PowerShell

```
aws iot test-invoke-authorizer --authorizer-name NAME_OF_AUTHORIZER  `
```

```
--mqtt-context '{"username": "USER_NAME", "password": "dGVzdA==",
 "clientId":"CLIENT_NAME"}'
```

The password must be base64-encoded. The following example shows how to encode a password in a Unix-like environment.

```
echo -n PASSWORD | base64
```

Managing custom authorizers

You can manage your authorizers by using the following APIs.

- ListAuthorizers: Show all authorizers in your account.
- DescribeAuthorizer: Displays properties of the specified authorizer. These values include creation date, last modified date, and other attributes.
- SetDefaultAuthorizer: Specifies the default authorizer for your AWS IoT Core data endpoints. AWS IoT Core uses this authorizer if a device doesn't pass AWS IoT Core credentials and doesn't specify an authorizer. For more information about using AWS IoT Core credentials, see the section called “Client authentication”.
- UpdateAuthorizer: Changes the status, token key name, or public keys for the specified authorizer.
- DeleteAuthorizer: Deletes the specified authorizer.

> **Note**
>
> You can't update an authorizer's signing requirement. This means that you can't disable signing in an existing authorizer that requires it. You also can't require signing in an existing authorizer that doesn't require it.

Custom authentication with X.509 client certificates

When connecting devices to AWS IoT Core, you have multiple authentication types available. You can use X.509 client certificates that can be used to authenticate client and device connections, or define custom authorizers to manage your own client authentication and authorization logic. This topic covers how to use custom authentication with X.509 client certificates.

Using custom authentication with X.509 certificates can be helpful if you've already authenticated your devices using X.509 certificates and want to perform additional validation and custom authorization. For example, if you store your devices' data such as their serial numbers in the X.509 client certificate, after AWS IoT Core authenticated the X.509 client certificate, you can use a custom authorizer to identify specific devices based on the information stored in the certificate's CommonName field. Using custom authentication with X.509 certificates can enhance your device security management when connecting devices to AWS IoT Core and provides more flexibility to manage the authentication and authorization logic. AWS IoT Core supports custom authentication with X.509 certificates using the X.509 certificate and custom authorizer authentication type, which works with both the MQTT protocol and the HTTPS protocol. For more information about the authentication types and application protocols that AWS IoT Core device endpoints support, see Device communication protocols.

Note

Custom authentication with X.509 client certificates is not supported in the AWS GovCloud (US) Regions.

Important

You must use an endpoint created using domain configurations. In addition, clients must provide the Server Name Indication (SNI) extension when connecting to AWS IoT Core.

The process to authenticate devices using custom authentication with X.509 client certificates consists of the following steps.

- Step 1: Register your X.509 client certificates with AWS IoT Core
- Step 2: Create a Lambda function
- Step 3: Create a custom authorizer
- Step 4: Set authentication type and application protocol in a domain configuration

Step 1: Register your X.509 client certificates with AWS IoT Core

If you haven't done this already, register and activate your X.509 client certificates with AWS IoT Core. Otherwise, skip to the next step.

To register and activate your client certificates with AWS IoT Core, follow the steps:

1. If you create client certificates directly with AWS IoT. These client certificates will be automatically registered with AWS IoT Core.
2. If you create your own client certificates, follow these instructions to register them with AWS IoT Core.
3. To activate your client certificates, follow these instructions.

Step 2: Create a Lambda function

AWS IoT Core uses custom authorizers to implement custom authentication and authorization schemes. A custom authorizer is associated with a Lambda function that determines whether a device is authenticated and what operations the device is allowed to perform. When a device connects to AWS IoT Core, AWS IoT Core retrieves the authorizer details including authorizer name and associated Lambda function, and invokes the Lambda function. The Lambda function receives an event that contains a JSON object with the device's X.509 client certificate data. Your Lambda function uses this event JSON object to evaluate the authentication request, decide the actions to take, and send a response back.

Lambda function event example

The following example JSON object contains all possible fields that can be included. The actual JSON object will only contain fields relevant to the specific connection request.

```
{
 "token": "aToken",
 "signatureVerified": true,
 "protocols": [
  "tls",
  "mqtt"
 ],
 "protocolData": {
  "tls": {
   "serverName": "serverName",
   "x509CertificatePem": "x509CertificatePem",
   "principalId": "principalId"
  },
  "mqtt": {
   "clientId": "myClientId",
                     "username": "myUserName",
```

```
                       "password": "myPassword"
   }
  },
  "connectionMetadata": {
   "id": "UUID"
  }
 }
```

`signatureVerified`

A Boolean value that indicates whether the token signature configured in the authorizer is verified or not before invoking the authorizer's Lambda function. If the authorizer is configured to disable token signing, this field will be false.

`protocols`

An array that contains the protocols to expect for the request.

`protocolData`

An object that contains information of the protocols used in the connection. It provides protocol-specific details that can be useful for authentication, authorization, and more.

`tls` - This object holds information related to the TLS (Transport Layer Security) protocol.

- `serverName` - The Server Name Indication (SNI) hostname string. AWS IoT Core requires devices to send the SNI extension to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field.
- `x509CertificatePem` - The X.509 certificate in PEM format, which is used for client authentication in the TLS connection.
- `principalId` - The principal identifier associated with the client in the TLS connection.

`mqtt` - This object holds information related to the MQTT protocol.

- `clientId` - A string only needs to be included in the event that the device sends this value.
- `username` - The user name provided in the MQTT Connect packet.
- `password` - The password provided in the MQTT Connect packet.

`connectionMetadata`

Metadata of the connection.

`id` - The connection ID, which you can use for logging and troubleshooting.

> **Note**
>
> In this event JSON object, `x509CertificatePem` and `principalId` are two new fields in the request. The value of `principalId` is the same as the value of `certificateId`. For more information, see Certificate.

Lambda function response example

The Lambda function should use information from the event JSON object to authenticate the incoming connection and decide what actions are permitted in the connection.

The following JSON object contains an example response that your Lambda function can send.

```
{
 "isAuthenticated": true,
 "principalId": "xxxxxxxx",
 "disconnectAfterInSeconds": 86400,
 "refreshAfterInSeconds": 300,
 "policyDocuments": [
  {
   "Version": "2012-10-17",
   "Statement": [
    {
     "Effect": "Allow",
     "Action": "iot:Publish",
     "Resource": "arn:aws:iot:us-east-1:123456789012:topic/customauthtesting"
    }
   ]
  }
 ]
}
```

In this example, this function should send a response that contains the following values.

`isAuthenticated`

A Boolean value that indicates whether the request is authenticated.

`principalId`

An alphanumeric string that acts as an identifier for the token sent by the custom authorization request. The value must be an alphanumeric string with at least one, and no more than 128, characters. It identifies the connection in logs. The value of `principalId` must be the same as the value of `principalId` in the event JSON object (i.e. certificateId of the X.509 certificate).

`policyDocuments`

A list of JSON-formatted AWS IoT Core policy documents. The value is optional and supports thing policy variables and certificate policy variables. The maximum number of policy documents is 10. Each policy document can contain a maximum of 2,048 characters. If you have multiple policies attached to your client certificate and the Lambda function, the permission is a collection of all policies. For more information about creating AWS IoT Core policies, see Policies.

`disconnectAfterInSeconds`

An integer that specifies the maximum duration (in seconds) of the connection to the AWS IoT Core gateway. The minimum value is 300 seconds, and the maximum value is 86,400 seconds. `disconnectAfterInSeconds` is for the lifetime of a connection and doesn't get refreshed on consecutive policy refreshes.

`refreshAfterInSeconds`

An integer that specifies the interval between policy refreshes. When this interval passes, AWS IoT Core invokes the Lambda function to allow for policy refreshes. The minimum value is 300 seconds, and the maximum value is 86,400 seconds.

Example Lambda function

The following is a sample Node.js Lambda function. The function examines the client's X.509 certificate and extracts relevant information such as the serial number, fingerprint, and subject name. If the extracted information matches the expected values, the client is granted access to connect. This mechanism ensures that only authorized clients with valid certificates can establish a connection.

```
const crypto = require('crypto');

exports.handler = async (event) => {

    // Extract the certificate PEM from the event
    const certPem = event.protocolData.tls.x509CertificatePem;

    // Parse the certificate using Node's crypto module
    const cert = new crypto.X509Certificate(certPem);
```

```
    var effect = "Deny";
    // Allow permissions only for a particular certificate serial, fingerprint, and
 subject
    if (cert.serialNumber === "7F8D2E4B9C1A5036DE8F7C4B2A91E5D80463BC9A1257" // This is
 a random serial
       && cert.fingerprint ===
 "F2:9A:C4:1D:B5:E7:08:3F:6B:D0:4E:92:A7:C1:5B:8D:16:0F:E3:7A" // This is a random
 fingerprint
       && cert.subject === "allow.example.com") {
      effect = "Allow";
    }

    return generateAuthResponse(event.protocolData.tls.principalId, effect);
};


// Helper function to generate the authorization response.
function generateAuthResponse(principalId, effect) {
    const authResponse = {
        isAuthenticated: true,
        principalId,
        disconnectAfterInSeconds: 3600,
        refreshAfterInSeconds: 300,
        policyDocuments: [
          {
            Version: "2012-10-17",
            Statement: [
              {
                Action: ["iot:Connect"],
                Effect: effect,
                Resource: [
                  "arn:aws:iot:us-east-1:123456789012:client/myClientName"
                ]
              },
              {
                Action: ["iot:Publish"],
                Effect: effect,
                Resource: [
                  "arn:aws:iot:us-east-1:123456789012:topic/telemetry/myClientName"
                ]
              },
              {
                Action: ["iot:Subscribe"],
```

```
                Effect: effect,
                Resource: [
                   "arn:aws:iot:us-east-1:123456789012:topicfilter/telemetry/
myClientName"
                ]
              },
              {
                Action: ["iot:Receive"],
                Effect: effect,
                Resource: [
                   "arn:aws:iot:us-east-1:123456789012:topic/telemetry/myClientName"
                ]
              }
            ]
          }
        ]
      };

  return authResponse;
}
```

The preceding Lambda function returns the following JSON when it receives a certificate with the expected serial, fingerprint, and subject. The value of `x509CertificatePem` will be the client certificate provided in the TLS handshake. For more information, see Defining your Lambda function.

```
{
 "isAuthenticated": true,
 "principalId": "principalId in the event JSON object",
 "policyDocuments": [
  {
   "Version": "2012-10-17",
   "Statement": [
    {
     "Action": "iot:Connect",
     "Effect": "Allow",
     "Resource": "arn:aws:iot:us-east-1:123456789012:client/myClientName"
    },
    {
     "Action": "iot:Publish",
     "Effect": "Allow",
     "Resource": "arn:aws:iot:us-east-1:123456789012:topic/telemetry/myClientName"
    },
```

```
    {
     "Action": "iot:Subscribe",
     "Effect": "Allow",
     "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/telemetry/
myClientName"
    },
    {
     "Action": "iot:Receive",
     "Effect": "Allow",
     "Resource": "arn:aws:iot:us-east-1:123456789012:topic/telemetry/myClientName"
    }
   ]
  }
 ],
 "disconnectAfterInSeconds": 3600,
 "refreshAfterInSeconds": 300
}
```

Step 3: Create a custom authorizer

After you define the Lambda function, create a custom authorizer to manage your own client authentication and authorization logic. You can follow the detailed instructions in Step 3: Create a customer authorizer resource and its authorization. For more information, see Creating an authorizer.

In the process of creating the custom authorizer, you must grant AWS IoT permission to invoke the Lambda function after it's created. For detailed instructions, see Authorizing AWS IoT to invoke your Lambda function.

Step 4: Set authentication type and application protocol in a domain configuration

To authenticate devices using custom authentication with X.509 client certificates, you must set the authentication type and application protocol in a domain configuration, and you must send the SNI extension. The value of `authenticationType` must be `CUSTOM_AUTH_X509`, and the value of `applicationProtocol` can either be `SECURE_MQTT` or `HTTPS`.

Set authentication type and application protocol in domain configuration (CLI)

If you don't have a domain configuration, use the **create-domain-configuration** command to create one. The value of `authenticationType` must be `CUSTOM_AUTH_X509`, and the value of `applicationProtocol` can either be `SECURE_MQTT` or `HTTPS`.

```
aws iot create-domain-configuration \
    --domain-configuration-name domainConfigurationName \
    --authentication-type CUSTOM_AUTH_X509 \
    --application-protocol SECURE_MQTT \
    --authorizer-config '{
        "defaultAuthorizerName": my-custom-authorizer
    }'
```

If you already have a domain configuration, use the **update-domain-configuration** command update `authenticationType` and `applicationProtocol` if needed. Note that you can't change the authentication type or protocol on the default endpoint (`iot:Data-ATS`).

```
aws iot update-domain-configuration \
    --domain-configuration-name domainConfigurationName \
    --authentication-type CUSTOM_AUTH_X509 \
    --application-protocol SECURE_MQTT \
    --authorizer-config '{
        "defaultAuthorizerName": my-custom-authorizer
    }'
```

`domain-configuration-name`

The name of the domain configuration.

`authentication-type`

The authentication type of the domain configuration. For more information, see choosing an authentication type.

`application-protocol`

The application protocol which devices use to communicate with AWS IoT Core. For more information, see choosing an application protocol.

`--authorizer-config`

An object that specifies the authorizer configuration in a domain configuration.

`defaultAuthorizerName`

The name of the authorizer for a domain configuration.

For more information, see CreateDomainConfiguration and UpdateDomainConfiguration from the *AWS IoT API Reference*. For more information about domain configuration, see Domain configurations.

Connecting to AWS IoT Core by using custom authentication

Devices can connect to AWS IoT Core by using custom authentication with any protocol that AWS IoT Core supports for device messaging. For more information about supported communication protocols, see the section called "Device communication protocols". The connection data that you pass to your authorizer Lambda function depends on the protocol you use. For more information about creating your authorizer Lambda function, see the section called "Defining your Lambda function". The following sections explain how to connect to authenticate by using each supported protocol.

HTTPS

Devices sending data to AWS IoT Core by using the HTTP Publish API can pass credentials either through request headers or query parameters in their HTTP POST requests. Devices can specify an authorizer to invoke by using the `x-amz-customauthorizer-name` header or query parameter. If you have token signing enabled in your authorizer, you must pass the *`token-key-name`* and `x-amz-customauthorizer-signature` in either request headers or query parameters. Note that the *`token-signature`* value must be URL-encoded when using JavaScript from within the browser.

> **Note**
>
> The customer authorizer for the HTTPS protocol only supports publish operations. For more information about the HTTPS protocol, see the section called "Device communication protocols".

The following example requests show how you pass these parameters in both request headers and query parameters.

```
//Passing credentials via headers
POST /topics/topic?qos=qos HTTP/1.1
Host: your-endpoint
x-amz-customauthorizer-signature: token-signature
token-key-name: token-value
x-amz-customauthorizer-name: authorizer-name
```

```
//Passing credentials via query parameters
POST /topics/topic?qos=qos&x-amz-customauthorizer-signature=token-signature&token-key-name=token-value HTTP/1.1
```

MQTT

Devices connecting to AWS IoT Core by using an MQTT connection can pass credentials through the `username` and `password` fields of MQTT messages. The `username` value can also optionally contain a query string that passes additional values (including a token, signature, and authorizer name) to your authorizer. You can use this query string if you want to use a token-based authentication scheme instead of `username` and `password` values.

> **Note**
>
> Data in the password field is base64-encoded by AWS IoT Core. Your Lambda function must decode it.

The following example contains a `username` string that contains extra parameters that specify a token and signature.

```
username?x-amz-customauthorizer-name=authorizer-name&x-amz-customauthorizer-signature=token-signature&token-key-name=token-value
```

To invoke an authorizer, devices connecting to AWS IoT Core by using MQTT and custom authentication must connect on port 443. They also must pass the Application Layer Protocol Negotiation (ALPN) TLS extension with a value of `mqtt` and the Server Name Indication (SNI) extension with the host name of their AWS IoT Core data endpoint. To avoid potential errors, the value for `x-amz-customauthorizer-signature` should be URL encoded. We also highly recommend that the values of `x-amz-customauthorizer-name` and `token-key-name` be URL encoded. For more information about these values, see the section called “Device communication protocols”. The V2 *AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client* can configure both of these extensions.

MQTT over WebSockets

Devices connecting to AWS IoT Core by using MQTT over WebSockets can pass credentials in one of the two following ways.

- Through request headers or query parameters in the HTTP UPGRADE request to establish the WebSockets connection.
- Through the `username` and `password` fields in the MQTT CONNECT message.

If you pass credentials through the MQTT connect message, the ALPN and SNI TLS extensions are required. For more information about these extensions, see the section called "MQTT". The following example demonstrates how to pass credentials through the HTTP Upgrade request.

```
GET /mqtt HTTP/1.1
Host: your-endpoint
Upgrade: WebSocket
Connection: Upgrade
x-amz-customauthorizer-signature: token-signature
token-key-name: token-value
sec-WebSocket-Key: any random base64 value
sec-websocket-protocol: mqtt
sec-WebSocket-Version: websocket version
```

Signing the token

You must sign the token with the private key of the public-private key pair that you used in the `create-authorizer` call. The following examples show how to create the token signature by using a UNIX-like command and JavaScript. They use the SHA-256 hash algorithm to encode the signature.

Command line

```
echo -n TOKEN_VALUE | openssl dgst -sha256 -sign PEM encoded RSA private key |
 openssl base64
```

JavaScript

```

const crypto = require('crypto')

const key = "PEM encoded RSA private key"

const k = crypto.createPrivateKey(key)
let sign = crypto.createSign('SHA256')
sign.write(t)
```

```
sign.end()
const s = sign.sign(k, 'base64')
```

Troubleshooting your authorizers

This topic walks through common issues that can cause problems in custom authentication workflows and steps for resolving them. To troubleshoot issues most effectively, enable CloudWatch logs for AWS IoT Core and set the log level to **DEBUG**. You can enable CloudWatch logs in the AWS IoT Core console (https://console.aws.amazon.com/iot/). For more information about enabling and configuring logs for AWS IoT Core, see the section called "Configure AWS IoT logging".

> **Note**
>
> If you leave the log level at **DEBUG** for long periods of time, CloudWatch might store large amounts of logging data. This can increase your CloudWatch charges. Consider using resource-based logging to increase the verbosity for only devices in a particular thing group. For more information about resource-based logging, see the section called "Configure AWS IoT logging". Also, when you're done troubleshooting, reduce the log level to a less verbose level.

Before you start troubleshooting, review the section called "Understanding the custom authentication workflow" for a high-level view of the custom authentication process. This helps you understand where to look for the source of a problem.

This topic discusses the following two areas for you to investigate.

- Issues related to your authorizer's Lambda function.
- Issues related to your device.

Check for issues in your authorizer's Lambda function

Perform the following steps to make sure that your devices' connection attempts are invoking your Lambda function.

1. Verify which Lambda function is associated with your authorizer.

   You can do this by calling the DescribeAuthorizer API or by clicking on the desired authorizer in the **Secure** section of the AWS IoT Core console.

2. Check the invocation metrics for the Lambda function. Perform the following steps to do this.

   a. Open the AWS Lambda console (https://console.aws.amazon.com/lambda/) and select the function that is associated with your authorizer.

   b. Choose the **Monitor** tab and view metrics for the time frame that is relevant to your problem.

3. If you see no invocations, verify that AWS IoT Core has permission to invoke your Lambda function. If you see invocations, skip to the next step. Perform the following steps to verify that your Lambda function has the required permissions.

   a. Choose the **Permissions** tab for your function in the AWS Lambda console.

   b. Find the **Resource-based Policy** section at the bottom of the page. If your Lambda function has the required permissions, the policy looks like the following example.

```
{
  "Version":"2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "Id123",
      "Effect": "Allow",
      "Principal": {
        "Service": "iot.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-
east-1:111111111111:function:FunctionName",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:iot:us-east-1:111111111111:authorizer/
AuthorizerName"
        },
        "StringEquals": {
          "AWS:SourceAccount": "111111111111"
        }
      }
```

```
    }
  ]
}
```

c. This policy grants the `InvokeFunction` permission on your function to the AWS IoT Core principal. If you don't see it, you'll have to add it by using the [AddPermission](#) API. The following example shows you how to do this by using the AWS CLI.

```
aws lambda add-permission --function-name FunctionName --principal
 iot.amazonaws.com --source-arn AuthorizerARn --statement-id Id-123 --action
 "lambda:InvokeFunction"
```

4. If you see invocations, verify that there are no errors. An error might indicate that the Lambda function isn't properly handling the connection event that AWS IoT Core sends to it.

   For information about handling the event in your Lambda function, see [the section called "Defining your Lambda function"](#). You can use the test feature in the AWS Lambda console (https://console.aws.amazon.com/lambda/) to hard-code test values in the function to make sure that the function is handling events correctly.

5. If you see invocations with no errors, but your devices are not able to connect (or publish, subscribe, and receive messages), the issue might be that the policy that your Lambda function returns doesn't give permissions for the actions that your devices are trying to take. Perform the following steps to determine whether anything is wrong with the policy that the function returns.

   a. Use an Amazon CloudWatch Logs Insights query to scan logs over a short period of time to check for failures. The following example query sorts events by timestamp and looks for failures.

```
display clientId, eventType, status, @timestamp | sort @timestamp desc | filter
 status = "Failure"
```

   b. Update your Lambda function to log the data that it's returning to AWS IoT Core and the event that triggers the function. You can use these logs to inspect the policy that the function creates.

6. If you see invocations with no errors, but your devices are not able to connect (or publish, subscribe, and receive messages), another reason can be that your Lambda function exceeds

the timeout limit. The Lambda function timeout limit for custom authorizer is 5 seconds. You can check the function duration in CloudWatch logs or metrics.

Investigating device issues

If you find no issues with invoking your Lambda function or with the policy that the function returns, look for problems with your devices' connection attempts. Malformed connection requests can cause AWS IoT Core not to trigger your authorizer. Connection problems can occur at both the TLS and application layers.

Possible TLS layer issues:

- Customers must pass either a hostname header (HTTP, MQTT over WebSockets) or the Server Name Indication TLS extension (HTTP, MQTT over WebSockets, MQTT) in all custom authentication requests. In both cases, the value passed must match one of your account's AWS IoT Core data endpoints. These are the endpoints that are returned when you perform the following CLI commands.
  - `aws iot describe-endpoint --endpoint-type iot:Data-ATS`
  - `aws iot describe-endpoint --endpoint-type iot:Data` (for legacy VeriSign endpoints)
- Devices that use custom authentication for MQTT connections must also pass the Application Layer Protocol Negotiation (ALPN) TLS extension with a value of `mqtt`.
- Custom authentication is currently available only on port 443.

Possible application layer issues:

- If signing is enabled (the `signingDisabled` field is false in your authorizer), look for the following signature issues.
  - Make sure that you're passing the token signature in either the `x-amz-customauthorizer-signature`header or in a query string parameter.
  - Make sure that the service isn't signing a value other than the token.
  - Make sure that you pass the token in the header or query parameter that you specified in the `token-key-name` field in your authorizer.
- Make sure that the authorizer name you pass in the `x-amz-customauthorizer-name` header or query string parameter is valid or that you have a default authorizer defined for your account.

Authorization

Authorization is the process of granting permissions to an authenticated identity. You grant permissions in AWS IoT Core using AWS IoT Core and IAM policies. This topic covers AWS IoT Core policies. For more information about IAM policies, see Identity and access management for AWS IoT and How AWS IoT works with IAM.

AWS IoT Core policies determine what an authenticated identity can do. An authenticated identity is used by devices, mobile applications, web applications, and desktop applications. An authenticated identity can even be a user typing AWS IoT Core CLI commands. An identity can execute AWS IoT Core operations only if it has a policy that grants it permission for those operations.

Both AWS IoT Core policies and IAM policies are used with AWS IoT Core to control the operations an identity (also called a *principal*) can perform. The policy type you use depends on the type of identity you are using to authenticate with AWS IoT Core.

AWS IoT Core operations are divided into two groups:

- Control plane API allows you to perform administrative tasks like creating or updating certificates, things, rules, and so on.
- Data plane API allows you send data to and receive data from AWS IoT Core.

The type of policy you use depends on whether you are using control plane or data plane API.

The following table shows the identity types, the protocols they use, and the policy types that can be used for authorization.

AWS IoT Core data plane API and policy types

Protocol and authentic ation mechanism	SDK	Identity type	Policy type		
MQTT over TLS/TCP, TLS mutual authentic	AWS IoT Device SDK	X.509 certificates	AWS IoT Core policy		

Protocol and authentication mechanism	SDK	Identity type	Policy type		
ation (port 8883 or 443)[‡])					
MQTT over HTTPS/WebSocket, AWS SigV4 authentication (port 443)	AWS Mobile SDK	Authenticated Amazon Cognito identity	IAM and AWS IoT Core policies		
		Unauthenticated Amazon Cognito identity	IAM policy		
		IAM, or federated identity	IAM policy		
HTTPS, AWS Signature Version 4 authentication (port 443)	AWS CLI	Amazon Cognito, IAM, or federated identity	IAM policy		
HTTPS, TLS mutual authentication (port 8443)	No SDK support	X.509 certificates	AWS IoT Core policy		

Protocol and authentication mechanism	SDK	Identity type	Policy type
HTTPS over custom authentication (Port 443)	AWS IoT Device SDK	Custom authorizer	Custom authorizer policy

AWS IoT Core control plane API and policy types

Protocol and authentication mechanism	SDK	Identity type	Policy type
HTTPS AWS Signature Version 4 authentication (port 443)	AWS CLI	Amazon Cognito identity	IAM policy
		IAM, or federated identity	IAM policy

AWS IoT Core policies are attached to X.509 certificates, Amazon Cognito identities, or thing groups. IAM policies are attached to an IAM user, group, or role. If you use the AWS IoT console or the AWS IoT Core CLI to attach the policy (to a certificate, Amazon Cognito Identity, or thing group), you use an AWS IoT Core policy. Otherwise, you use an IAM policy. AWS IoT Core policies attached to a thing group applies to any thing within that thing group. For the AWS IoT Core policy to take effect, the `clientId` and the thing name must match.

Policy-based authorization is a powerful tool. It gives you complete control over what a device, user, or application can do in AWS IoT Core. For example, consider a device connecting to AWS IoT Core with a certificate. You can allow the device to access all MQTT topics, or you can restrict its access to a single topic. In another example, consider a user typing CLI commands at the command line. By using a policy, you can allow or deny access to any command or AWS IoT Core resource for the user. You can also control an application's access to AWS IoT Core resources.

Changes made to a policy can take a few minutes to become effective because of how AWS IoT caches the policy documents. That is, it may take a few minutes to access a resource that has recently been granted access, and a resource may be accessible for several minutes after its access has been revoked.

AWS training and certification

For information about authorization in AWS IoT Core, take the Deep Dive into AWS IoT Core Authentication and Authorization course on the AWS Training and Certification website.

AWS IoT Core policies

AWS IoT Core policies are JSON documents. They follow the same conventions as IAM policies. AWS IoT Core supports named policies so many identities can reference the same policy document. Named policies are versioned so they can be easily rolled back.

AWS IoT Core policies allow you to control access to the AWS IoT Core data plane. The AWS IoT Core data plane consists of operations that allow you to connect to the AWS IoT Core message broker, send and receive MQTT messages, and get or update a thing's Device Shadow.

An AWS IoT Core policy is a JSON document that contains one or more policy statements. Each statement contains:

- `Effect`, which specifies whether the action is allowed or denied.
- `Action`, which specifies the action the policy is allowing or denying.
- `Resource`, which specifies the resource or resources on which the action is allowed or denied.

Changes made to a policy can take anywhere between 6 and 8 minutes to become effective because of how AWS IoT caches the policy documents. That is, it may take a few minutes to access a resource that has recently been granted access, and a resource may be accessible for several minutes after its access has been revoked.

AWS IoT Core policies can be attached to X.509 certificates, Amazon Cognito identities, and thing groups. The policies attached to a thing group apply to any thing within that group. For the policy to take effect, the `clientId` and the thing name must match. AWS IoT Core policies follow the same policy evaluation logic as IAM policies. By default, all policies are implicitly denied. An explicit allow in any identity-based or resource-based policy overrides the default behavior. An explicit deny in any policy overrides any allows. For more information, see Policy evaluation logic in the *AWS Identity and Access Management User Guide*.

Topics

AWS IoT Core policy actions

The following policy actions are defined by AWS IoT Core:

MQTT Policy Actions

`iot:Connect`

Represents the permission to connect to the AWS IoT Core message broker. The `iot:Connect` permission is checked every time a `CONNECT` request is sent to the broker. The message broker doesn't allow two clients with the same client ID to stay connected at the same time. After the second client connects, the broker closes the existing connection. Use the `iot:Connect` permission to ensure only authorized clients using a specific client ID can connect.

`iot:DeleteConnection`

Represents the permission to disconnect a connected MQTT client from AWS IoT Core. The `iot:DeleteConnection` permission is checked every time a request is made to forcibly disconnect a client. When you disconnect a client, AWS IoT Core closes the client's network connection and optionally cleans the session state.

`iot:GetRetainedMessage`

Represents the permission to get the contents of a single retained message. Retained messages are the messages that were published with the RETAIN flag set and stored by AWS IoT Core. For permission to get a list of all the account's retained messages, see `iot:ListRetainedMessages`.

`iot:ListRetainedMessages`

Represents the permission to retrieve summary information about the account's retained messages, but not the contents of the messages. Retained messages are the messages that were published with the RETAIN flag set and stored by AWS IoT Core. The resource ARN specified for this action must be `*`. For permission to get the contents of a single retained message, see `iot:GetRetainedMessage`.

`iot:Publish`

Represents the permission to publish an MQTT topic. This permission is checked every time a PUBLISH request is sent to the broker. You can use this to allow clients to publish to specific topic patterns.

> **Note**
>
> To grant `iot:Publish` permission, you must also grant `iot:Connect` permission.

`iot:Receive`

Represents the permission to receive a message from AWS IoT Core. The `iot:Receive` permission is confirmed every time a message is delivered to a client. Because this permission is checked on every delivery, you can use it to revoke permissions to clients that are currently subscribed to a topic.

`iot:RetainPublish`

Represents the permission to publish an MQTT message with the RETAIN flag set.

> **Note**
>
> To grant `iot:RetainPublish` permission, you must also grant `iot:Publish` permission.

`iot:Subscribe`

Represents the permission to subscribe to a topic filter. This permission is checked every time a SUBSCRIBE request is sent to the broker. Use it to allow clients to subscribe to topics that match specific topic patterns.

> **Note**
>
> To grant `iot:Subscribe` permission, you must also grant `iot:Connect` permission.

Device Shadow Policy Actions

`iot:DeleteThingShadow`

Represents the permission to delete a thing's Device Shadow. The `iot:DeleteThingShadow` permission is checked every time a request is made to delete a thing's Device Shadow contents.

`iot:GetThingShadow`

Represents the permission to retrieve a thing's Device Shadow. The `iot:GetThingShadow` permission is checked every time a request is made to retrieve a thing's Device Shadow contents.

`iot:ListNamedShadowsForThing`

Represents the permission to list a thing's named Shadows. The `iot:ListNamedShadowsForThing` permission is checked every time a request is made to list a thing's named Shadows.

`iot:UpdateThingShadow`

Represents the permission to update a device's shadow. The `iot:UpdateThingShadow` permission is checked every time a request is made to update a thing's Device Shadow contents.

> **Note**
>
> The job execution policy actions apply only for the HTTP TLS endpoint. If you use the MQTT endpoint, you must use MQTT policy actions defined in this topic.
> For an example of a job execution policy that demonstrates this, see the section called "Basic job policy example" that works with the MQTT protocol.

Job Executions AWS IoT Core Policy Actions

`iotjobsdata:DescribeJobExecution`

Represents the permission to retrieve a job execution for a given thing. The `iotjobsdata:DescribeJobExecution` permission is checked every time a request is made to get a job execution.

`iotjobsdata:GetPendingJobExecutions`

Represents the permission to retrieve the list of jobs that are not in a terminal status for a thing. The `iotjobsdata:GetPendingJobExecutions` permission is checked every time a request is made to retrieve the list.

`iotjobsdata:UpdateJobExecution`

Represents the permission to update a job execution. The `iotjobsdata:UpdateJobExecution` permission is checked every time a request is made to update the state of a job execution.

`iotjobsdata:StartNextPendingJobExecution`

Represents the permission to get and start the next pending job execution for a thing. (That is, to update a job execution with status QUEUED to IN_PROGRESS.) The `iotjobsdata:StartNextPendingJobExecution` permission is checked every time a request is made to start the next pending job execution.

AWS IoT Core Credential Provider Policy Action

`iot:AssumeRoleWithCertificate`

Represents the permission to call AWS IoT Core credential provider to assume an IAM role with certificate-based authentication. The `iot:AssumeRoleWithCertificate` permission is checked every time a request is made to AWS IoT Core credential provider to assume a role.

AWS IoT Core action resources

To specify a resource for an AWS IoT Core policy action, use the Amazon Resource Name (ARN) of the resource. All resource ARNs follow the following format:

```
arn:partition:iot:region:AWS-account-ID:Resource-type/Resource-name
```

The following table shows the resource to specify for each action type. The ARN examples are for the account ID 123456789012, in the partition aws, and specific to the region us-east-1. For more information about the formats for ARNs, see Amazon Resource Names (ARNs) from the AWS Identity and Access Management User Guide.

Action	Resource type	Resource name	ARN example
iot:Connect	client	The client's client ID	arn:aws:iot:us-east-1:123456789012:client/myClientId
iot:DeleteConnection	client	The client's client ID	arn:aws:iot:us-east-1:123456789012:client/myClientId
iot:DeleteThingShadow	thing	The thing's name, and the shadow's name, if applicable	arn:aws:iot:us-east-1:123456789012:thing/thingOne arn:aws:iot:us-east-1:123456789012:thing/thingOne/shadowOne
iotjobsdata:DescribeJobExecution	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne
iotjobsdata:GetPendingJobExecutions	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:GetRetainedMessage	topic	A retained message topic	arn:aws:iot:us-east-1:123456789012:topic/myTopicName

Action	Resource type	Resource name	ARN example
iot:GetThingShadow	thing	The thing's name, and the shadow's name, if applicable	arn:aws:iot:us-east-1:123456789012:thing/thingOne arn:aws:iot:us-east-1:123456789012:thing/thingOne/shadowOne
iot:ListNamedShadowsForThing	All	All	*
iot:ListRetainedMessages	All	All	*
iot:Publish	topic	A topic string	arn:aws:iot:us-east-1:123456789012:topic/myTopicName
iot:Receive	topic	A topic string	arn:aws:iot:us-east-1:123456789012:topic/myTopicName
iot:RetainPublish	topic	A topic to publish with the RETAIN flag set	arn:aws:iot:us-east-1:123456789012:topic/myTopicName
iotjobsdata:StartNextPendingJobExecution	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne

Action	Resource type	Resource name	ARN example
iot:Subscribe	topicfilter	A topic filter string	arn:aws:iot:us-east-1:123456789012:topicfilter/myTopicFilter
iotjobsdata:UpdateJobExecution	thing	The thing's name	arn:aws:iot:us-east-1:123456789012:thing/thingOne
iot:UpdateThingShadow	thing	The thing's name, and the shadow's name, if applicable	arn:aws:iot:us-east-1:123456789012:thing/thingOne arn:aws:iot:us-east-1:123456789012:thing/thingOne/shadowOne
iot:AssumeRoleWithCertificate	rolealias	A role alias that points to a role ARN	arn:aws:iot:us-east-1:123456789012:rolealias/CredentialProviderRole_alias

AWS IoT Core policy variables

AWS IoT Core defines policy variables that can be used in AWS IoT Core policies in the `Resource` or `Condition` block. When a policy is evaluated, the policy variables are replaced by actual values. For example, if a device is connected to the AWS IoT Core message broker with a client ID of 100-234-3456, the `iot:ClientId` policy variable is replaced in the policy document by 100-234-3456.

AWS IoT Core policies can use wildcard characters and follow a similar convention to IAM policies. Inserting an * (asterik) in the string can be treated as a wildcard, matching any characters. For example, you can use * to describe multiple MQTT topic names in the `Resource` attribute of a policy. The characters + and # are treated as literal strings in a policy. For an example policy that shows how to use wildcards, see [Using wildcard characters in MQTT and AWS IoT Core policies](#).

You can also use predefined policy variables with fixed values to represent characters that otherwise have special meaning. These special characters include `$(*)`, `$(?)`, and `$($)`. For more information about policy variables and the special characters, see IAM Policy elements: Variables and tags and Creating a condition with multiple keys or values.

Topics

- Basic AWS IoT Core policy variables
- Thing policy variables
- X.509 Certificate AWS IoT Core policy variables

Basic AWS IoT Core policy variables

AWS IoT Core defines the following basic policy variables:

- `aws:SourceIp`: The IP address of the client connected to the AWS IoT Core message broker.
- `iot:ClientId`: The client ID used to connect to the AWS IoT Core message broker.
- `iot:DomainName`: The domain name of the client connected to AWS IoT Core.

Examples

- Examples of ClientId and SourceIp policy variables
- Examples of iot:DomainName policy variable

Examples of `ClientId` and `SourceIp` policy variables

The following AWS IoT Core policy shows a policy that uses policy variables. `aws:SourceIp` can be used in the Condition element of your policy to allow principals to make API requests only within a specific address range. For examples, see Authorizing users and cloud services to use AWS IoT Jobs.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
```

```
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/clientid1"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/my/topic/${iot:ClientId}"
      ],
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": "123.45.167.89"
        }
      }
    }
  ]
}
```

In these examples, `${iot:ClientId}` is replaced by the ID of the client connected to the AWS IoT Core message broker when the policy is evaluated. When you use policy variables like `${iot:ClientId}`, you can inadvertently open access to unintended topics. For example, if you use a policy that uses `${iot:ClientId}` to specify a topic filter:

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Subscribe"
  ],
  "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topicfilter/my/${iot:ClientId}/topic"
  ]
}
```

A client can connect using + as the client ID. This would allow the user to subscribe to any topic that matches the topic filter `my/+/topic`. To protect against such security gaps, use the `iot:Connect` policy action to control which client IDs can connect. For example, this policy allows only those clients whose client ID is `clientid1` to connect:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/clientid"
   ]
  }
 ]
}
```

> **Note**
>
> Using the policy variable `${iot:ClientId}` with `Connect` is not recommended. There is no check on the value of `ClientId`, so an attacher with a different client's ID can pass the validation but cause disconnection. Because any `ClientId` is allowed, setting a random client ID can bypass thing group policies.

Examples of `iot:DomainName` policy variable

You can add the `iot:DomainName` policy variable to restrict which domains are allowed to use. Adding the `iot:DomainName` policy variable allows devices to connect to only specific configured endpoints.

The following policy allows devices to connect to the specified domain.

```
{
 "Version":"2012-10-17",
 "Statement": {
  "Sid": "AllowConnectionsToSpecifiedDomain",
  "Effect": "Allow",
  "Action": [
   "iot:Connect"
```

```
  ],
  "Resource": "arn:aws:iot:us-east-1:123456789012:client/clientid",
  "Condition": {
   "StringEquals": {
    "iot:DomainName": "d1234567890abcdefghij-ats.iot.us-east-1.amazonaws.com"
   }
  }
 }
}
```

The following policy denies devices to connect to the specified domain.

```
{
 "Version":"2012-10-17",
 "Statement": {
  "Sid": "DenyConnectionsToSpecifiedDomain",
  "Effect": "Deny",
  "Action": [
   "iot:Connect"
  ],
  "Resource": "arn:aws:iot:us-east-1:123456789012:client/clientid",
  "Condition": {
   "StringEquals": {
    "iot:DomainName": "d1234567890abcdefghij-ats.iot.us-east-1.amazonaws.com"
   }
  }
 }
}
```

For more information about policy conditional operator, see IAM JSON policy elements: Condition operators. For more information about domain configurations, see What is a domain configuration?.

Thing policy variables

Thing policy variables allow you to write AWS IoT Core policies that grant or deny permissions based on thing properties like thing names, thing types, and thing attribute values. You can use thing policy variables to apply the same policy to control many AWS IoT Core devices. For more information about device provisioning, see Device Provisioning.

If you use non-exclusive thing association, the same certificate can be attached to multiple things. To maintain a clear association and to avoid potential conflicts, you must match your client ID with the thing name. In this case, you obtain the thing name from the client ID in the MQTT `Connect` message sent when a thing connects to AWS IoT Core.

Keep the following in mind when using thing policy variables in AWS IoT Core policies.

- Use the [AttachThingPrincipal](#) API to attach certificates or principals (authenticated Amazon Cognito identities) to a thing.
- If non-exclusive thing association is in place, when you're replacing thing names with thing policy variables, the value of `clientId` in the MQTT connect message or the TLS connection must exactly match the thing name.

The following thing policy variables are available:

- `iot:Connection.Thing.ThingName`

  This resolves to the name of the thing in the AWS IoT Core registry for which the policy is being evaluated. AWS IoT Core uses the certificate the device presents when it authenticates to determine which thing to use to verify the connection. This policy variable is only available when a device connects over MQTT or MQTT over the WebSocket protocol.
- `iot:Connection.Thing.ThingTypeName`

  This resolves to the thing type associated with the thing for which the policy is being evaluated. The client ID of the MQTT/WebSocket connection must be the same as the thing name. This policy variable is available only when connecting over MQTT or MQTT over the WebSocket protocol.
- `iot:Connection.Thing.Attributes[`*`attributeName`*`]`

  This resolves to the value of the specified attribute associated with the thing for which the policy is being evaluated. A thing can have up to 50 attributes. Each attribute is available as a policy variable: `iot:Connection.Thing.Attributes[`*`attributeName`*`]` where *`attributeName`* is the name of the attribute. The client ID of the MQTT/WebSocket connection must be the same as the thing name. This policy variable is only available when connecting over MQTT or MQTT over the WebSocket protocol.
- `iot:Connection.Thing.IsAttached`

`iot:Connection.Thing.IsAttached: ["true"]` enforces that only the devices that are both registered in AWS IoT and attached to principal can access the permissions inside the policy. You can use this variable to prevent a device from connecting to AWS IoT Core if it presents a certificate that is not attached to an IoT thing in the AWS IoT Core registry.This variable has values `true` or `false` indicating that the connecting thing is attached to the certificate or Amazon Cognito identity in the registry using AttachThingPrincipal API. Thing name is taken as client Id.

If your client ID matches your thing name, or if you attach your certificate to a thing exclusively, using policy variables in the policy definition can simplify policy management. Instead of creating individual policies for each IoT thing, you can define a single policy using the thing policy variables. This policy can be applied to all devices dynamically. The following is an example policy to show how it works. For more information, see ???.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Condition": {
    "StringLike": {
     "iot:ClientId": "*${iot:Connection.Thing.Attributes[envType]}"
    }
   },
   "Effect": "Allow",
   "Action": "iot:Connect",
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/*"
  }
 ]
}
```

This policy example allows things to connect to AWS IoT Core if their client ID ends with the value of their `envType` attribute. Only things with a matching client ID pattern will be allowed to connect.

X.509 Certificate AWS IoT Core policy variables

X.509 certificate policy variables assist with writing AWS IoT Core policies. These policies grant permissions based on X.509 certificate attributes. The following sections describe how to use these certificate policy variables.

> **Important**
>
> If your X.509 certificate doesn't include a particular certificate attribute but the corresponding certificate policy variable is used in your policy document, the policy evaluation might lead to unexpected behavior.

CertificateId

In the RegisterCertificate API, the `certificateId` appears in the response body. To get information about your certificate, use the `certificateId` in DescribeCertificate.

Issuer attributes

The following AWS IoT Core policy variables support the allowing or denying of permissions, based on certificate attributes set by the certificate issuer.

- `iot:Certificate.Issuer.DistinguishedNameQualifier`
- `iot:Certificate.Issuer.Country`
- `iot:Certificate.Issuer.Organization`
- `iot:Certificate.Issuer.OrganizationalUnit`
- `iot:Certificate.Issuer.State`
- `iot:Certificate.Issuer.CommonName`
- `iot:Certificate.Issuer.SerialNumber`
- `iot:Certificate.Issuer.Title`
- `iot:Certificate.Issuer.Surname`
- `iot:Certificate.Issuer.GivenName`
- `iot:Certificate.Issuer.Initials`
- `iot:Certificate.Issuer.Pseudonym`

- `iot:Certificate.Issuer.GenerationQualifier`

Subject attributes

The following AWS IoT Core policy variables support the granting or denying of permissions, based on certificate subject attributes set by the certificate issuer.

- `iot:Certificate.Subject.DistinguishedNameQualifier`
- `iot:Certificate.Subject.Country`
- `iot:Certificate.Subject.Organization`
- `iot:Certificate.Subject.OrganizationalUnit`
- `iot:Certificate.Subject.State`
- `iot:Certificate.Subject.CommonName`
- `iot:Certificate.Subject.SerialNumber`
- `iot:Certificate.Subject.Title`
- `iot:Certificate.Subject.Surname`
- `iot:Certificate.Subject.GivenName`
- `iot:Certificate.Subject.Initials`
- `iot:Certificate.Subject.Pseudonym`
- `iot:Certificate.Subject.GenerationQualifier`

X.509 certificates provide these attributes with the option to contain one or more values. By default, the policy variables for each multi-value attribute return the first value. For example, the `Certificate.Subject.Country` attribute might contain a list of country names, but when evaluated in a policy, `iot:Certificate.Subject.Country` is replaced by the first country name.

You can request a specific attribute value other than the first value by using a one-based index. For example, `iot:Certificate.Subject.Country.1` is replaced by the second country name in the `Certificate.Subject.Country` attribute. If you specify an index value that does not exist (for example, if you ask for a third value when there are only two values assigned to the attribute), no substitution is made and authorization fails. You can use the `.List` suffix on the policy variable name to specify all values of the attribute.

Issuer alternate name attributes

The following AWS IoT Core policy variables support the granting or denying of permissions, based on issuer alternate name attributes set by the certificate issuer.

- `iot:Certificate.Issuer.AlternativeName.RFC822Name`
- `iot:Certificate.Issuer.AlternativeName.DNSName`
- `iot:Certificate.Issuer.AlternativeName.DirectoryName`
- `iot:Certificate.Issuer.AlternativeName.UniformResourceIdentifier`
- `iot:Certificate.Issuer.AlternativeName.IPAddress`

Subject alternate name attributes

The following AWS IoT Core policy variables support the granting or denying of permissions, based on subject alternate name attributes set by the certificate issuer.

- `iot:Certificate.Subject.AlternativeName.RFC822Name`
- `iot:Certificate.Subject.AlternativeName.DNSName`
- `iot:Certificate.Subject.AlternativeName.DirectoryName`
- `iot:Certificate.Subject.AlternativeName.UniformResourceIdentifier`
- `iot:Certificate.Subject.AlternativeName.IPAddress`

Other attributes

You can use `iot:Certificate.SerialNumber` to allow or deny access to AWS IoT Core resources, based on the serial number of a certificate. The `iot:Certificate.AvailableKeys` policy variable contains the name of all certificate policy variables that contain values.

Using X.509 certificate policy variables

This topic provides details of how to use certificate policy variables. X.509 certificate policy variables are essential when you create AWS IoT Core policies that give permissions based on X.509 certificate attributes. If your X.509 certificate doesn't include a particular certificate attribute but the corresponding certificate policy variable is used in your policy document, the policy evaluation might lead to unexpected behavior. This is because the missing policy variable doesn't get evaluated in the policy statement.

In this topic:

X.509 certificate example

A typical X.509 certificate might appear as follows. This example certificate includes certificate attributes. During the evaluation of AWS IoT Core policies, the following certificate attributes will be populated as certificate policy variables: `Serial Number`, `Issuer`, `Subject`, `X509v3 Issuer Alternative Name`, and `X509v3 Subject Alternative Name`.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            92:12:85:cb:b7:a5:e0:86
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=IoT Devices, OU=SmartHome, ST=WA, CN=IoT Devices Primary CA,
    GN=Primary CA1/initials=XY/dnQualifier=Example corp,
    SN=SmartHome/ title=CA1/pseudonym=Primary_CA/generationQualifier=2/serialNumber=987

        Validity
            Not Before: Mar 26 03:25:40 2024 GMT
            Not After : Apr 28 03:25:40 2025 GMT
        Subject: C=US, O=IoT Devices, OU=LightBulb, ST=NY, CN=LightBulb Device Cert,
    GN=Bulb/initials=ZZ/dnQualifier=Bulb001,
    SN=Multi Color/title=RGB/pseudonym=RGB Device/generationQualifier=4/
serialNumber=123
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
```

```
                    << REDACTED >>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Subject Alternative Name:
                DNS:example.com, IP Address:1.2.3.4, URI:ResourceIdentifier001,
email:device1@example.com, DirName:/C=US/O=IoT/OU=SmartHome/CN=LightBulbCert
            X509v3 Issuer Alternative Name:
                DNS:issuer.com, IP Address:5.6.7.8, URI:PrimarySignerCA,
email:primary@issuer.com, DirName:/C=US/O=Issuer/OU=IoT Devices/CN=Primary Issuer CA
   Signature Algorithm: sha256WithRSAEncryption
        << REDACTED >>
```

Using certificate issuer attributes as certificate policy variables

The following table provides details of how certificate issuer attributes will be populated in an AWS IoT Core policy.

Issuer attributes to be populated in a policy

Certificate issuer attributes	Certificate policy variables
• C=US • O=IoT Devices • OU=SmartHome • ST=WA • CN=IoT Devices Primary CA • GN=Primary CA1 • initials=XY • dnQualifier=Example corp • SN=SmartHome • title=CA1 • pseudonym=Primary_CA • generationQualifier=2 • serialNumber=987	• `iot:Certificate.Issuer.Country = US` • `iot:Certificate.Issuer.Organization = IoT Devices` • `iot:Certificate.Issuer.OrganizationalUnit = SmartHome` • `iot:Certificate.Issuer.State = WA` • `iot:Certificate.Issuer.CommonName = IoT Devices Primary CA` • `iot:Certificate.Issuer.GivenName = Primary CA1` • `iot:Certificate.Issuer.initials = XY` • `iot:Certificate.Issuer.DistinguishedNameQualifier = Example corp` • `iot:Certificate.Issuer.Surname = SmartHome`

Certificate issuer attributes	Certificate policy variables
	• `iot:Certificate.Issuer.Title = CA1` • `iot:Certificate.Issuer.Pseudonym = Primary_CA` • `iot:Certificate.Issuer.GenerationQualifier = 2` • `iot:Certificate.Issuer.SerialNumber = 987`

Using certificate subject attributes as certificate policy variables

The following table provides details of how certificate subject attributes will be populated in an AWS IoT Core policy.

Subject attributes to be populated in a policy

Certificate subject attributes	Certificate policy variables
• C=US • O=IoT Devices • ST=NY • CN=LightBulb Device Cert • GN=Bulb • initials=ZZ • dnQualifier=Bulb001 • SN=Multi Color • title=RGB • pseudonym=RGB Device • generationQualifier=4 • serialNumber=123	• `iot:Certificate.Subject.Country = US` • `iot:Certificate.Subject.Organization = IoT Devices` • `iot:Certificate.Subject.State = NY` • `iot:Certificate.Subject.CommonName = LightBulb Device Cert` • `iot:Certificate.Subject.GivenName = Bulb` • `iot:Certificate.Subject.initials = ZZ` • `iot:Certificate.Subject.DistinguishedNameQualifier = Bulb001` • `iot:Certificate.Subject.Surname = Multi Color` • `iot:Certificate.Subject.Title = RGB` • `iot:Certificate.Subject.Pseudonym = RGB Device` • `iot:Certificate.Subject.GenerationQualifier = 4`

Certificate subject attributes	Certificate policy variables
	• `iot:Certificate.Subject.SerialNumber = 123`

Using certificate Issuer alternate name attributes as certificate policy variables

The following table provides details of how certificate issuer alternate name attributes will be populated in an AWS IoT Core policy.

Issuer alternate name attributes to be populated in a policy

X509v3 Issuer Alternative Name	Attribute in a policy
• DNS:issuer.com • IP Address:5.6.7.8 • URI:PrimarySignerCA • email:primary@issuer.com • DirName:/C=US/O=Issuer/OU=IoT Devices/CN=Primary Issuer CA	• `iot:Certificate.Issuer.AlternativeName.DNSName = issuer.com` • `iot:Certificate.Issuer.AlternativeName.IPAddress = 5.6.7.8` • `iot:Certificate.Issuer.AlternativeName.UniformResourceIdentifier = PrimarySignerCA` • `iot:Certificate.Issuer.AlternativeName.RFC822Name = primary@issuer.com` • `iot:Certificate.Issuer.AlternativeName.DirectoryName = cn=Primary Issuer CA,ou=IoT Devices,o=Issuer,c=US`

Using certificate subject alternate name attributes as certificate policy variables

The following table provides details of how certificate subject alternate name attributes will be populated in an AWS IoT Core policy.

Subject alternate name attributes to be populated in a policy

X509v3 Subject Alternative Name	Attribute in a policy
• DNS:example.com • IP Address:1.2.3.4 • URI:ResourceIdentifier001 • email:device1@example.com • DirName:/C=US/O=IoT/OU=SmartHome/CN=LightBulbCert	• `iot:Certificate.Subject.AlternativeName.DNSName = example.com` • `iot:Certificate.Subject.AlternativeName.IPAddress = 1.2.3.4` • `iot:Certificate.Subject.AlternativeName.UniformResourceIdentifier = ResourceIdentifier001` • `iot:Certificate.Subject.AlternativeName.RFC822Name = device1@example.com` • `iot:Certificate.Subject.AlternativeName.DirectoryName = cn=LightBulbCert,ou=SmartHome,o=IoT,c=US`

Using other certificate attribute as a certificate policy variable

The following table provides details of how other certificate attributes will be populated in an AWS IoT Core policy.

Other attributes to be populated in a policy

Other certificate attribute	Certificate policy variable
`Serial Number: 92:12:85:cb:b7:a5:e0:86`	`iot:Certificate.SerialNumber = 10525622389124227206`

X.509 Certificate policy variable limitations

The following limitations apply to X.509 certificate policy variables:

Missing policy variables

If your X.509 certificate doesn't include a particular certificate attribute but the corresponding certificate policy variable is used in your policy document, the policy evaluation might lead to unexpected behavior. This is because the missing policy variable doesn't get evaluated in the policy statement.

Certificate SerialNumber format

AWS IoT Core treats the certificate serial number as the string representation of a decimal integer. For example, if a policy only allows connections with Client ID matching the certificate serial number, the client ID must be the serial number in decimal format.

Wildcards

If wildcard characters are present in certificate attributes, the policy variable is not replaced by the certificate attribute value. This will leave the `${policy-variable}` text in the policy document. This might cause authorization failure. The following wildcard characters can be used: *, $, +, ?, and #.

Array fields

Certificate attributes that contain arrays are limited to five items. Additional items are ignored.

String length

All string values are limited to 1024 characters. If a certificate attribute contains a string longer than 1024 characters, the policy variable is not replaced by the certificate attribute value. This will leave the `${policy-variable}` in the policy document. This might cause authorization failure.

Special Characters

Any special character, such as `,`, ", \, +, =, <, > and `;` must be prefixed with a backslash (\) when used in a policy variable. For example, `Amazon Web Services O=Amazon.com Inc. L=Seattle ST=Washington C=US` becomes `Amazon Web Service O\=Amazon.com Inc. L\=Seattle ST\=Washington C\=US`.

Example policies using certificate policy variables

The following policy document allows connections with client ID that matches the certificate serial number and publishing to the topic that matches the pattern: `${iot:Certificate.Subject.Organization}/device-stats/${iot:ClientId}/*`.

> **Important**
>
> If your X.509 certificate doesn't include a particular certificate attribute but the corresponding certificate policy variable is used in your policy document, the policy evaluation might lead to unexpected behavior. This is because the missing policy variable doesn't get evaluated in the policy statement. For example, if you attach the following policy document to a certificate that doesn't contain the `iot:Certificate.Subject.Organization` attribute, the `iot:Certificate.Subject.Organization` certificate policy variables won't be populated during the policy evaluation.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Certificate.SerialNumber}"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Subject.Organization}/device-stats/${iot:ClientId}/*"
   ]
  }
 ]
}
```

You can also use the Null condition operator to ensure that the certificate policy variables used in a policy are populated during policy evaluation. The following policy document allows `iot:Connect` with certificates only when the Certificate Serial Number and Certificate Subject Common name attributes are present.

All of the certificate policy variables have String values, so all of the [String condition operators](#) are supported.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/*"
   ],
   "Condition": {
    "Null": {
     "iot:Certificate.SerialNumber": "false",
     "iot:Certificate.Subject.CommonName": "false"
    }
   }
  }
 ]
}
```

Cross-service confused deputy prevention

The *confused deputy problem* is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it shouldn't otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

To limit the permissions that AWS IoT gives another service to the resource, we recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies.

If you use both global condition context keys, the aws:SourceAccount value and the account in the aws:SourceArn value must use the same account ID when used in the same policy statement.

The most effective way to protect against the confused deputy problem is to use the aws:SourceArn global condition context key with the full Amazon Resource Name (ARN) of the resource. For AWS IoT, your aws:SourceArn must comply with the format: arn:*aws*:iot:*region*:*account-id*:*resource-type/resource-id* for resource specific permissions or arn:*aws*:iot:*region*:*account-id*:*. The resource-id can be the name or ID of the permitted resource, or a wildcard statement of the permitted resource IDs. Make sure that the *region* matches your AWS IoT Region and the *account-id* matches your customer account ID.

The following example shows how to prevent the confused deputy problem by using the aws:SourceArn and aws:SourceAccount global condition context keys in the AWS IoT role trust policy. For more examples, see Detailed examples of confused deputy prevention.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "Service":"iot.amazonaws.com"
         },
         "Action":"sts:AssumeRole",
         "Condition":{
            "StringEquals":{
               "aws:SourceAccount":"123456789012"
        },
            "ArnLike":{
               "aws:SourceArn":"arn:aws:iot:us-east-1:123456789012:*"
        }
         }
      }
   ]
}
```

> **Note**
>
> If you get access deny errors, it can be because the service integration with AWS Security Token Service (STS) doesn't support the `aws:SourceArn` and `aws:SourceAccount` context keys.

Detailed examples of confused deputy prevention

This section provides detailed examples of how to prevent the confused deputy problem by using the aws:SourceArn and aws:SourceAccount global condition context keys in the AWS IoT role trust policy.

- Fleet provisioning
- JITP
- Credential provider

Fleet provisioning

You can configure fleet provisioning using a provisioning template resource. When a provisioning template references a provisioning role, that role's trust policy can include the `aws:SourceArn` and `aws:SourceAccount` condition keys. These keys limit the resources for which the configuration can invoke the `sts:AssumeRole` request.

The role with the following trust policy can only be assumed by the IoT principal (`iot.amazonaws.com`) for the provisioning template specified in the `SourceArn`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "Service":"iot.amazonaws.com"
         },
         "Action":"sts:AssumeRole",
         "Condition":{
            "StringEquals":{
               "aws:SourceAccount":"123456789012"
```

```
        },
            "ArnLike":{
               "aws:SourceArn":"arn:aws:iot:us-
east-1:123456789012:provisioningtemplate/example_template"
        }
         }
      }
   ]
}
```

JITP

In just-in-time provisioning (JITP), you can either use provisioning template as a resource separate from the CA or define the template body and the role as part of the CA certificate configuration. The value of `aws:SourceArn` in the AWS IoT role trust policy depends on how you define the provisioning template.

Defining provisioning template as a separate resource

If you define your provisioning template as a separate resource, the value of `aws:SourceArn` can be `"arn:aws:iot:region:account-id:provisioningtemplate/example_template"`. You can use the same policy example in Fleet provisioning.

Defining provisioning template in a CA certificate

If you define your provisioning template within a CA certificate resource, the value of `aws:SourceArn` can be `"arn:aws:iot:region:account-id:cacert/cert_id"` or `"arn:aws:iot:region:account-id:cacert/*"`. You can use a wildcard when the resource identifier, such as the ID of a CA certificate, is unknown at the time of creation.

The role with the following trust policy can only be assumed by the IoT principal (`iot.amazonaws.com`) for the CA certificate specified in the `SourceArn`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "Service":"iot.amazonaws.com"
```

```
            },
            "Action":"sts:AssumeRole",
            "Condition":{
               "StringEquals":{
                  "aws:SourceAccount":"123456789012"
           },
               "ArnLike":{
                  "aws:SourceArn":"arn:aws:iot:us-
east-1:123456789012:cacert/
8ecde6884f3d87b1125ba31ac3fcb13d7016de7f57cc904fe1cb97c6ae98196e"
           }
            }
         }
      ]
}
```

When creating a CA certificate, you can reference a provisioning role in the registration configuration. The trust policy of the provisioning role can use `aws:SourceArn` to restrict what resources the role can be assumed for. However, during the initial [RegisterCACertificate](https://docs.aws.amazon.com/iot/latest/apireference/API_RegisterCACertificate.html) call to register the CA certificate, you would not have the ARN of the CA certificate to specify in the `aws:SourceArn` condition.

To work around this, i.e., to specify the provisioning role trust policy to the specific CA certificate that's registered with AWS IoT Core, you can do the following:

- First, call [RegisterCACertificate](https://docs.aws.amazon.com/iot/latest/apireference/API_RegisterCACertificate.html) without providing the `RegistrationConfig` parameter.
- After the CA certificate is registered with AWS IoT Core, call [UpdateCACertificate](https://docs.aws.amazon.com/iot/latest/apireference/API_UpdateCACertificate.html) on it.

  In the UpdateCACertificate call, provide a `RegistrationConfig` that includes the provisioning role trust policy with `aws:SourceArn` set to the ARN of the newly registered CA certificate.

Credential provider

For [AWS IoT Core credential provider](https://docs.aws.amazon.com/iot/latest/developerguide/authorizing-direct-aws.html), use the same AWS account you use to create the role alias in `aws:SourceAccount`, and specify a statement that matches the resource ARN of the rolealias resource type in `aws:SourceArn`. When creating an IAM role for use with AWS IoT Core credential provider, you must include in the `aws:SourceArn` condition the ARNs of any role aliases that might need to assume the role, thereby authorizing the cross-service `sts:AssumeRole` request.

The role with the following trust policy can only be assumed by the principal of AWS IoT Core credential provider (`credentials.iot.amazonaws.com`) for the roleAlias specified in the `SourceArn`. If a principal attempts to retrieve credentials for a role alias other than what's specified in the `aws:SourceArn` condition, the request will be denied, even if that other role alias references the same IAM role.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:iot:us-
east-1:123456789012:rolealias/example_rolealias"
        }
      }
    }
  ]
}
```

AWS IoT Core policy examples

The example policies in this section illustrate the policy documents used to complete common tasks in AWS IoT Core. You can use them as examples to start from when creating the policies for your solutions.

The examples in this section use these policy elements:

- the section called “AWS IoT Core policy actions”
- the section called “AWS IoT Core action resources”
- the section called “Identity-based policy examples”

- the section called "Basic AWS IoT Core policy variables"
- the section called "X.509 Certificate AWS IoT Core policy variables"

Policy examples in this section:

- Connect policy examples
- Publish/Subscribe policy examples
- Connect and publish policy examples
- Retained message policy examples
- Certificate policy examples
- Thing policy examples
- Basic job policy example

Connect policy examples

The following policy denies permission to client IDs `client1` and `client2` to connect to AWS IoT Core, while allowing devices to connect using a client ID. The client ID matches the name of a thing that's registered in the AWS IoT Core registry and attached to the principal that's used for connection:

> **Note**
>
> For registered devices, we recommend that you use thing policy variables for `Connect` actions and attach the thing to the principal that's used for the connection.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Deny",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/client1",
    "arn:aws:iot:us-east-1:123456789012:client/client2"
```

```
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  }
 ]
}
```

The following policy grants permission to connect to AWS IoT Core with client ID `client1`. This policy example is for unregistered devices.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/client1"
   ]
  }
 ]
}
```

MQTT persistent sessions policy examples

`connectAttributes` allow you to specify what attributes you want to use in your connect message in your IAM policies such as `PersistentConnect` and `LastWill`. For more information, see Using connectAttributes.

The following policy allows connect with `PersistentConnect` feature:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
   "Condition": {
    "ForAllValues:StringEquals": {
     "iot:ConnectAttributes": [
      "PersistentConnect"
     ]
    }
   }
  }
 ]
}
```

The following policy disallows `PersistentConnect`, other features are allowed:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
   "Condition": {
```

```
    "ForAllValues:StringNotEquals": {
     "iot:ConnectAttributes": [
      "PersistentConnect"
     ]
    }
   }
  }
 ]
}
```

The above policy can also be expressed using `StringEquals`, any other feature including new feature is allowed:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1"
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
            }
        }
        }
    ]
}
```

The following policy allows connect by both `PersistentConnect` and `LastWill`, any other new feature is not allowed:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
   "Condition": {
    "ForAllValues:StringEquals": {
     "iot:ConnectAttributes": [
      "PersistentConnect",
      "LastWill"
     ]
    }
   }
  }
 ]
}
```

The following policy allows clean connect by clients with or without `LastWill`, no other features will be allowed:

```
{
    "Version":"2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
        ],
        "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
        "Condition": {
            "StringEquals": {
                "iot:ConnectAttributes": "LastWill"
        }
        }
```

```
    }]
}
```

The following policy only allows connect using default features:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
   "Condition": {
    "ForAllValues:StringEquals": {
     "iot:ConnectAttributes": []
    }
   }
  }
 ]
}
```

The following policy allows connect only with `PersistentConnect`, any new feature is allowed as long as the connection uses `PersistentConnect`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
```

```
                    ]
                }
            }
        }
    ]
}
```

The following policy states the connect must have both `PersistentConnect` and `LastWill` usage, no new feature is allowed:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
   "Condition": {
    "ForAllValues:StringEquals": {
     "iot:ConnectAttributes": [
      "PersistentConnect",
      "LastWill"
     ]
    }
   }
  },
  {
   "Effect": "Deny",
   "Action": [
    "iot:Connect"
   ],
   "Resource": "*",
   "Condition": {
    "ForAllValues:StringEquals": {
     "iot:ConnectAttributes": [
      "PersistentConnect"
     ]
    }
   }
```

```
   },
   {
    "Effect": "Deny",
    "Action": [
     "iot:Connect"
    ],
    "Resource": "*",
    "Condition": {
     "ForAllValues:StringEquals": {
      "iot:ConnectAttributes": [
       "LastWill"
      ]
     }
    }
   },
   {
    "Effect": "Deny",
    "Action": [
     "iot:Connect"
    ],
    "Resource": "*",
    "Condition": {
     "ForAllValues:StringEquals": {
      "iot:ConnectAttributes": []
     }
    }
   }
  ]
}
```

The following policy must not have `PersistentConnect` but can have `LastWill`, any other new feature is not allowed:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
```

```
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
            }
        }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "iot:ConnectAttributes": [
                        "LastWill"
                    ]
            }
        }
        }
    ]
}
```

The following policy allows connect only by clients that have a `LastWill` with topic `"my/lastwill/topicName"`, any feature is allowed as long as it uses the `LastWill` topic:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
            "Condition": {
                "ArnEquals": {
```

```
                "iot:LastWillTopic": "arn:aws:iot:us-east-1:123456789012:topic/my/
lastwill/topicName"
            }
        }
        }
    ]
}
```

The following policy only allows clean connect using a specific `LastWillTopic`, any feature is allowed as long as it uses the `LastWillTopic`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/client1",
            "Condition": {
                "ArnEquals": {
                "iot:LastWillTopic": "arn:aws:iot:us-east-1:123456789012:topic/my/
lastwill/topicName"
            }
        }
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:ConnectAttributes": [
                        "PersistentConnect"
                    ]
            }
        }
        }
```

```
    ]
}
```

Publish/Subscribe policy examples

The policy you use depends on how you're connecting to AWS IoT Core. You can connect to AWS IoT Core by using an MQTT client, HTTP, or WebSocket. When you connect with an MQTT client, you're authenticating with an X.509 certificate. When you connect over HTTP or the WebSocket protocol, you're authenticating with Signature Version 4 and Amazon Cognito.

> **Note**
>
> For registered devices, we recommend that you use thing policy variables for `Connect` actions and attach the thing to the principal that's used for the connection.

In this section:

- Using wildcard characters in MQTT and AWS IoT Core policies
- Policies to publish, subscribe and receive messages to/from specific topics
- Policies to publish, subscribe and receive messages to/from topics with a specific prefix
- Policies to publish, subscribe and receive messages to/from topics specific to each device
- Policies to publish, subscribe and receive messages to/from topics with thing attribute in topic name
- Policies to deny publishing messages to subtopics of a topic name
- Policies to deny receiving messages from subtopics of a topic name
- Policies to subscribe to topics using MQTT wildcard characters
- Policies for HTTP and WebSocket clients

Using wildcard characters in MQTT and AWS IoT Core policies

MQTT and AWS IoT Core policies have different wildcard characters and you should choose them after careful consideration. In MQTT, the wildcard characters + and # are used in MQTT topic filters to subscribe to multiple topic names. AWS IoT Core policies use * and ? as wildcard characters and follow the conventions of IAM policies. In a policy document, the * represents any combination of characters and a question mark ? represents any single character. In policy documents, the

MQTT wildcard characters, + and # are treated as those characters with no special meaning. To describe multiple topic names and topic filters in the `resource` attribute of a policy, use the `*` and `?` wildcard characters in place of the MQTT wildcard characters.

When you choose the wildcard characters to use in a policy document, consider that the `*` character is not confined to a single topic level. The `+` character is confined to a single topic level in an MQTT topic filter. To help constrain a wildcard specification to a single MQTT topic filter level, consider using multiple `?` characters. For more information about using wildcard characters in a policy resource and more examples of what they match, see [Using wildcards in resource ARNs](#).

The table below shows the different wildcard characters used in MQTT and AWS IoT Core policies for MQTT clients.

Wildcard character	Is MQTT wildcard character	Example in MQTT	Is AWS IoT Core policy wildcard character	Example in AWS IoT Core policies for MQTT clients
#	Yes	`some/#`	No	N/A
+	Yes	`some/+/topic`	No	N/A
*	No	N/A	Yes	`topicfilter/some/*/topic` `topicfilter/some/sensor*/topic`
?	No	N/A	Yes	`topic/some/?????/topic` `topicfilter/some/sensor???/topic`

Policies to publish, subscribe and receive messages to/from specific topics

The following shows examples for registered and unregistered devices to publish, subscribe and receive messages to/from the topic named "some_specific_topic". The examples also highlight that `Publish` and `Receive` use "topic" as the resource, and `Subscribe` uses "topicfilter" as the resource.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. It also provides `Publish`, `Subscribe` and `Receive` permissions for the topic named "some_specific_topic".

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/some_specific_topic"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Subscribe"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topicfilter/some_specific_topic"
   ]
  },
  {
   "Effect": "Allow",
```

```
    "Action": [
     "iot:Receive"
    ],
    "Resource": [
     "arn:aws:iot:us-east-1:123456789012:topic/some_specific_topic"
    ]
   }
  ]
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. It also provides `Publish`, `Subscribe` and `Receive` permissions for the topic named "some_specific_topic".

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/some_specific_topic"
            ]
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/
some_specific_topic"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/some_specific_topic"
            ]
        }
    ]
}
```

Policies to publish, subscribe and receive messages to/from topics with a specific prefix

The following shows examples for registered and unregistered devices to publish, subscribe and receive messages to/from topics prefixed with "topic_prefix".

> **Note**
>
> Note the use of the wildcard character * in this example. Although * is useful to provide permissions for multiple topic names in a single statement, it can lead to unintended consequences by providing more privileges to devices than required. So we recommend that you only use the wildcard character * after careful consideration.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. It also provides `Publish`, `Subscribe` and `Receive` permissions for topics prefixed with "topic_prefix".

```
{
```

```
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish",
    "iot:Receive"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/topic_prefix*"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Subscribe"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topicfilter/topic_prefix*"
   ]
  }
 ]
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. It also provides `Publish`, `Subscribe` and `Receive` permissions for topics prefixed with "topic_prefix".

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/topic_prefix*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/topic_prefix*"
            ]
        }
    ]
}
```

Policies to publish, subscribe and receive messages to/from topics specific to each device

The following shows examples for registered and unregistered devices to publish, subscribe and receive messages to/from topics that are specific to the given device.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. It provides permission to publish to the thing-specific topic (`sensor/device/${iot:Connection.Thing.ThingName}`) and also subscribe to and receive from the thing-specific topic (`command/device/${iot:Connection.Thing.ThingName}`). If the thing name in the registry is "thing1", the device will be able to publish to the topic "sensor/device/thing1". The device will also be able to subscribe to and receive from the topic "command/device/thing1".

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/sensor/device/
${iot:Connection.Thing.ThingName}"
   ]
  },
```

```
  {
   "Effect": "Allow",
   "Action": [
    "iot:Subscribe"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topicfilter/command/device/
${iot:Connection.Thing.ThingName}"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Receive"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/command/device/
${iot:Connection.Thing.ThingName}"
   ]
  }
 ]
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. It provides permission to publish to the client-specific topic (`sensor/device/${iot:ClientId}`), and also subscribe to and receive from the client-specific topic (`command/device/${iot:ClientId}`). If the device connects with clientId as clientId1, it will be able to publish to the topic "sensor/device/clientId1". The device will also be able to subscribe to and receive from the topic `device/clientId1/command`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
```

```
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/sensor/device/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/command/device/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/command/device/
${iot:Connection.Thing.ThingName}"
            ]
        }
    ]
}
```

Policies to publish, subscribe and receive messages to/from topics with thing attribute in topic name

The following shows an example for registered devices to publish, subscribe and receive messages to/from topics whose names include thing attributes.

> **Note**
>
> Thing attributes only exist for devices registered in AWS IoT Core Registry. There is no corresponding example for unregistered devices.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. It provides permission to publish to the topic (`sensor/${iot:Connection.Thing.Attributes[version]}`), and subscribe to and receive from the topic (`command/${iot:Connection.Thing.Attributes[location]}`) where the topic name includes thing attributes. If the thing name in the registry has `version=v1` and `location=Seattle`, the device will be able to publish to the topic "sensor/v1", and subscribe to and receive from the topic "command/Seattle".

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
```

```
    {
     "Effect": "Allow",
     "Action": [
      "iot:Publish"
     ],
     "Resource": [
      "arn:aws:iot:us-east-1:123456789012:topic/sensor/
${iot:Connection.Thing.Attributes[version]}"
     ]
    },
    {
     "Effect": "Allow",
     "Action": [
      "iot:Subscribe"
     ],
     "Resource": [
      "arn:aws:iot:us-east-1:123456789012:topicfilter/command/
${iot:Connection.Thing.Attributes[location]}"
     ]
    },
    {
     "Effect": "Allow",
     "Action": [
      "iot:Receive"
     ],
     "Resource": [
      "arn:aws:iot:us-east-1:123456789012:topic/command/
${iot:Connection.Thing.Attributes[location]}"
     ]
    }
  ]
}
```

Unregistered devices

Because thing attributes only exist for devices registered in AWS IoT Core registry, there is no corresponding example for unregistered things.

Policies to deny publishing messages to subtopics of a topic name

The following shows examples for registered and unregistered devices to publish messages to all topics except certain subtopics.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. It provides permission to publish to all topics prefixed with "department/" but not to the "department/admins" subtopic.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/department/*"
   ]
  },
  {
   "Effect": "Deny",
   "Action": [
    "iot:Publish"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:topic/department/admins"
   ]
  }
 ]
```

```
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. It provides permission to publish to all topics prefixed with "department/" but not to the "department/admins" subtopic.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/department/*"
            ]
        },
        {
            "Effect": "Deny",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/department/admins"
            ]
        }
    ]
```

```
}
```

Policies to deny receiving messages from subtopics of a topic name

The following shows examples for registered and unregistered devices to subscribe to and receive messages from topics with specific prefixes except certain subtopics.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. The policy allows devices to subscribe to any topic prefixed with "topic_prefix". By using `NotResource` in the statement for `iot:Receive`, we allow the device to receive messages from all topics that the device has subscribed to, except the topics prefixed with "topic_prefix/restricted". For example, with this policy, devices can subscribe to "topic_prefix/topic1" and even "topic_prefix/restricted", however, they will only receive messages from the topic "topic_prefix/topic1" and no messages from the topic "topic_prefix/restricted".

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ],
   "Condition": {
    "Bool": {
     "iot:Connection.Thing.IsAttached": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": "iot:Subscribe",
   "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/topic_prefix/*"
  },
```

```
  {
   "Effect": "Allow",
   "Action": "iot:Receive",
   "NotResource": "arn:aws:iot:us-east-1:123456789012:topic/topic_prefix/
restricted/*"
  }
 ]
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. The policy allows devices to subscribe to any topic prefixed with "topic_prefix". By using `NotResource` in the statement for `iot:Receive`, we allow the device to receive messages from all topics that the device has subscribed to, except topics prefixed with "topic_prefix/restricted". For example, with this policy, devices can subscribe to "topic_prefix/topic1" and even "topic_prefix/restricted". However, they will only receive messages from the topic "topic_prefix/topic1" and no messages from the topic "topic_prefix/restricted".

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/
topic_prefix/*"
        },
```

```
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
            "NotResource": "arn:aws:iot:us-east-1:123456789012:topic/
topic_prefix/restricted/*"
        }
    ]
}
```

Policies to subscribe to topics using MQTT wildcard characters

MQTT wildcard characters + and # are treated as literal strings, but they are not treated as wildcards when used in AWS IoT Core policies. In MQTT, + and # are treated as wildcards only when subscribing to a topic filter but as a literal string in all other contexts. We recommend that you only use these MQTT wildcards as part of AWS IoT Core policies after careful consideration.

The following shows examples for registered and unregistered things using MQTT wildcards in AWS IoT Core policies. These wildcards are treated as literal strings.

Registered devices

For devices registered in AWS IoT Core registry, the following policy allows devices to connect with clientId that matches the name of a thing in the registry. The policy allows devices to subscribe to the topics "department/+/employees" and "location/#". Because + and # are treated as literal strings in AWS IoT Core policies, devices can subscribe to the topic "department/+/employees" but not to the topic "department/engineering/employees". Similarly, devices can subscribe to the topic "location/#" but not to the topic "location/Seattle". However, once the device subscribes to the topic "department/+/employees", the policy will allow them to receive messages from the topic "department/engineering/employees". Similarly, once the device subscribes to the topic "location/#", they will receive messages from the topic "location/Seattle" as well.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
```

```
    ],
    "Resource": [
     "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
    ],
    "Condition": {
     "Bool": {
      "iot:Connection.Thing.IsAttached": "true"
     }
    }
   },
   {
    "Effect": "Allow",
    "Action": "iot:Subscribe",
    "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/department/+/
employees"
   },
   {
    "Effect": "Allow",
    "Action": "iot:Subscribe",
    "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/location/#"
   },
   {
    "Effect": "Allow",
    "Action": "iot:Receive",
    "Resource": "arn:aws:iot:us-east-1:123456789012:topic/*"
   }
  ]
}
```

Unregistered devices

For devices not registered in AWS IoT Core registry, the following policy allows devices to connect using either clientId1, clientId2 or clientId3. The policy allows devices to subscribe to the topics of "department/+/employees" and "location/#". Because + and # are treated as literal strings in AWS IoT Core policies, devices can subscribe to the topic "department/+/employees" but not to the topic "department/engineering/employees". Similarly, devices can subscribe to the topic "location/#" but not "location/Seattle". However, once the device subscribes to the topic "department/+/employees", the policy will allow them to receive messages from the topic "department/engineering/employees". Similarly, once the device subscribes to the topic "location/#", they will receive messages from the topic "location/Seattle" as well.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/clientId1",
                "arn:aws:iot:us-east-1:123456789012:client/clientId2",
                "arn:aws:iot:us-east-1:123456789012:client/clientId3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/
department/+/employees"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topicfilter/location/
#"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topic/*"
        }
    ]
}
```

Policies for HTTP and WebSocket clients

When you connect over HTTP or the WebSocket protocol, you're authenticating with Signature Version 4 and Amazon Cognito. Amazon Cognito identities can be authenticated or unauthenticated. Authenticated identities belong to users who are authenticated by any supported identity provider. Unauthenticated identities typically belong to guest users who do not authenticate with an identity provider. Amazon Cognito provides a unique identifier and AWS credentials to support unauthenticated identities. For more information, see the section called "Authorization with Amazon Cognito identities".

For the following operations, AWS IoT Core uses AWS IoT Core policies attached to Amazon Cognito identities through the `AttachPolicy` API. This scopes down the permissions attached to the Amazon Cognito Identity pool with authenticated identities.

- `iot:Connect`
- `iot:Publish`
- `iot:Subscribe`
- `iot:Receive`
- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

That means an Amazon Cognito Identity needs permission from the IAM role policy and the AWS IoT Core policy. You attach the IAM role policy to the pool and the AWS IoT Core policy to the Amazon Cognito Identity through the AWS IoT Core `AttachPolicy` API.

Authenticated and unauthenticated users are different identity types. If you don't attach an AWS IoT policy to the Amazon Cognito Identity, an authenticated user fails authorization in AWS IoT and doesn't have access to AWS IoT resources and actions.

> **Note**
>
> For other AWS IoT Core operations or for unauthenticated identities, AWS IoT Core does not scope down the permissions attached to the Amazon Cognito identity pool role. For both authenticated and unauthenticated identities, this is the most permissive policy that we recommend you attach to the Amazon Cognito pool role.

HTTP

To allow unauthenticated Amazon Cognito identities to publish messages over HTTP on a topic specific to the Amazon Cognito Identity, attach the following IAM policy to the Amazon Cognito Identity pool role:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/${cognito-
identity.amazonaws.com:sub}"
            ]
        }
    ]
}
```

To allow authenticated users, attach the preceding policy to the Amazon Cognito Identity pool role and to the Amazon Cognito Identity using the AWS IoT Core AttachPolicy API.

> **Note**
>
> When authorizing Amazon Cognito identities, AWS IoT Core considers both policies and grants the least privileges specified. An action is allowed only if both policies allow the requested action. If either policy disallows an action, that action is unauthorized.

MQTT

To allow unauthenticated Amazon Cognito identities to publish MQTT messages over WebSocket on a topic specific to the Amazon Cognito Identity in your account, attach the following IAM policy to the Amazon Cognito Identity pool role:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
```

```
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/${cognito-identity.amazonaws.com:sub}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/${cognito-identity.amazonaws.com:sub}"]
        }
    ]
}
```

To allow authenticated users, attach the preceding policy to the Amazon Cognito Identity pool role and to the Amazon Cognito Identity using the AWS IoT Core AttachPolicy API.

> **Note**
>
> When authorizing Amazon Cognito identities, AWS IoT Core considers both and grants the least privileges specified. An action is allowed only if both policies allow the requested action. If either policy disallows an action, that action is unauthorized.

Connect and publish policy examples

For devices registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name and restricts the device to publishing on a client-ID or thing name-specific MQTT topic. For a connection to be successful, the thing name must be registered in the AWS IoT Core registry and be authenticated using an identity or principal attached to the thing:

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
```

```
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/${iot:Connection.Thing.ThingName}"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
      ]
    }
  ]
}
```

For devices not registered as things in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client ID `client1` and restricts the device to publishing on a clientID-specific MQTT topic:

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/${iot:ClientId}"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/client1"
```

```
      ]
    }
  ]
}
```

Retained message policy examples

Using [retained messages](#) requires specific policies. Retained messages are MQTT messages published with the RETAIN flag set and stored by AWS IoT Core. This section presents examples of policies that allow common uses of retained messages.

In this section:

- Policy to connect and publish retained messages
- Policy to connect and publish retained Will messages
- Policy to list and get retained messages

Policy to connect and publish retained messages

For a device to publish retained messages, the device must be able to connect, publish (any MQTT message), and publish MQTT retained messages. The following policy grants these permissions for the topic: `device/sample/configuration` to client **`device1`**. For another example that grants permission to connect, see the section called "Connect and publish policy examples".

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "iot:Connect"
   ],
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/device1"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish",
```

```
        "iot:RetainPublish"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/device/sample/configuration"
      ]
    }
  ]
}
```

Policy to connect and publish retained Will messages

Clients can configure a message that AWS IoT Core will publish when the client disconnects unexpectedly. MQTT calls such a message a *Will* message. A client must have an additional condition added to its connect permission to include them.

The following policy document grants all clients permission to connect and publish a Will message, identified by its topic, `will`, that AWS IoT Core will also retain.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/device1"
      ],
      "Condition": {
        "ForAllValues:StringEquals": {
          "iot:ConnectAttributes": [
            "LastWill"
          ]
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
```

```
        "iot:RetainPublish"
       ],
       "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/will"
       ]
      }
     ]
    }
```

Policy to list and get retained messages

Services and applications can access retained messages without the need to support an MQTT client by calling `ListRetainedMessages` and `GetRetainedMessage`. The services and applications that call these actions must be authorized by using a policy such as the following example.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:ListRetainedMessages"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/device1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetRetainedMessage"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/foo"
            ]
        }
    ]
}
```

Certificate policy examples

For devices registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches a thing name, and to publish to a topic whose name is equal to the `certificateId` of the certificate the device used to authenticate itself:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:CertificateId}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
        }
    ]
}
```

For devices not registered in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs, `client1`, `client2`, and `client3` and to publish to a topic whose name is equal to the `certificateId` of the certificate the device used to authenticate itself:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
```

```
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:CertificateId}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        }
    ]
}
```

For devices registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name, and to publish to a topic whose name is equal to the subject's `CommonName` field of the certificate the device used to authenticate itself:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Subject.CommonName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
```

```
        }
    ]
}
```

> **Note**
>
> In this example, the certificate's subject common name is used as the topic identifier, with the assumption that the subject common name is unique for each registered certificate. If the certificates are shared across multiple devices, the subject common name is the same for all the devices that share this certificate, thereby allowing publish privileges to the same topic from multiple devices (not recommended).

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs, `client1`, `client2`, and `client3` and to publish to a topic whose name is equal to the subject's `CommonName` field of the certificate the device used to authenticate itself:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/
${iot:Certificate.Subject.CommonName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
```

```
        }
    ]
}
```

> **Note**
>
> In this example, the certificate's subject common name is used as the topic identifier, with the assumption that the subject common name is unique for each registered certificate. If the certificates are shared across multiple devices, the subject common name is the same for all the devices that share this certificate, thereby allowing publish privileges to the same topic from multiple devices (not recommended).

For devices registered in the AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with a client ID that matches the thing name, and to publish to a topic whose name is prefixed with `admin/` when the certificate used to authenticate the device has its `Subject.CommonName.2` field set to `Administrator`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/*"],
            "Condition": {
                "StringEquals": {
                    "iot:Certificate.Subject.CommonName.2": "Administrator"
            }
```

```
        }
        }
    ]
}
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs `client1`, `client2`, and `client3` and to publish to a topic whose name is prefixed with `admin/` when the certificate used to authenticate the device has its `Subject.CommonName.2` field set to `Administrator`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/*"],
            "Condition": {
                "StringEquals": {
                    "iot:Certificate.Subject.CommonName.2": "Administrator"
            }
        }
        }
    ]
}
```

For devices registered in AWS IoT Core registry, the following policy allows a device to use its thing name to publish on a specific topic that consists of `admin/` followed by the `ThingName` when the certificate used to authenticate the device has any one of its `Subject.CommonName` fields set to `Administrator`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin/
${iot:Connection.Thing.ThingName}"],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:Certificate.Subject.CommonName.List": "Administrator"
            }
        }
        }
    ]
}
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to connect to AWS IoT Core with client IDs `client1`, `client2`, and `client3` and to publish to the topic `admin` when the certificate used to authenticate the device has any one of its `Subject.CommonName` fields set to `Administrator`:

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1",
                "arn:aws:iot:us-east-1:123456789012:client/client2",
                "arn:aws:iot:us-east-1:123456789012:client/client3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/admin"],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "iot:Certificate.Subject.CommonName.List": "Administrator"
            }
        }
        }
    ]
}
```

Thing policy examples

The following policy allows a device to connect if the certificate used to authenticate with AWS IoT Core is attached to the thing for which the policy is being evaluated:

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Effect":"Allow",
            "Action":["iot:Connect"],
            "Resource":[ "*" ],
            "Condition": {
                "Bool": {
                    "iot:Connection.Thing.IsAttached": ["true"]
```

```
                }
            }
            }
        ]
}
```

The following policy allows a device to publish if the certificate is attached to a thing with a particular thing type and if the thing has an attribute of `attributeName` with value `attributeValue`. For more information about thing policy variables, see Thing policy variables.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": "arn:aws:iot:us-east-1:123456789012:topic/device/stats",
      "Condition": {
        "StringEquals": {
          "iot:Connection.Thing.Attributes[attributeName]": "attributeValue",
          "iot:Connection.Thing.ThingTypeName": "Thing_Type_Name"
        },
        "Bool": {
          "iot:Connection.Thing.IsAttached": "true"
        }
      }
    }
  ]
}
```

The following policy allows a device to publish to a topic that starts with an attribute of the thing. If the device certificate is not associated with the thing, this variable won't be resolved and will result in an access denied error. For more information about thing policy variables, see Thing policy variables.

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.Attributes[attributeName]}/*"
    }
  ]
}
```

Basic job policy example

This sample shows the policy statments required for a job target that's a single device to receive a job request and communicate job execution status with AWS IoT.

Replace *us-west-2:57EXAMPLE833* with your AWS Region, a colon character (:), and your 12-digit AWS account number, and then replace *uniqueThingName* with the name of the thing resource that represents the device in AWS IoT.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:client/uniqueThingName"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/test/dc/pubtopic",
```

```
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/events/job/*",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/events/jobExecution/
*",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topicfilter/test/dc/subtopic",
                "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/events/
jobExecution/*",
                "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/test/dc/subtopic",
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iotjobsdata:DescribeJobExecution",
                "iotjobsdata:GetPendingJobExecutions",
                "iotjobsdata:StartNextPendingJobExecution",
                "iotjobsdata:UpdateJobExecution"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/
things/uniqueThingName"
            ]
        }
```

```
    ]
}
```

Authorization with Amazon Cognito identities

There are two types of Amazon Cognito identities: unauthenticated and authenticated. If your app supports unauthenticated Amazon Cognito identities, no authentication is performed, so you don't know who the user is.

Unauthenticated Identities: For unauthenticated Amazon Cognito identities, you grant permissions by attaching an IAM role to an unauthenticated identity pool. We recommend that you only grant access to those resources you want available to unknown users.

> **Important**
>
> For unauthenticated Amazon Cognito users connecting to AWS IoT Core, we recommend that you give access to very limited resources in IAM policies.

Authenticated Identities: For authenticated Amazon Cognito identities, you need to specify permissions in two places:

- Attach an IAM policy to the authenticated Amazon Cognito Identity pool and
- Attach an AWS IoT Core policy to the Amazon Cognito Identity (authenticated user).

Policy examples for unauthenticated and authenticated Amazon Cognito users connecting to AWS IoT Core

The following example shows permissions in both the IAM policy and the IoT policy of an Amazon Cognito identity. The authenticated user wants to publish to a device specific topic (e.g. device/DEVICE_ID/status).

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/Client_ID"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/device/Device_ID/status"
            ]
        }
    ]
}
```

The following example shows the permissions in an IAM policy of an Amazon Cognito unauthenticated role. The unauthenticated user wants to publish to non-device specific topics that do not require authentication.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
```

```
                "Resource": [
                    "arn:aws:iot:us-east-1:123456789012:topic/non_device_specific_topic"
                ]
            }
        ]
}
```

GitHub examples

The following example web applications on GitHub show how to incorporate policy attachment to authenticated users into the user signup and authentication process.

- MQTT publish/subscribe React web application using AWS Amplify and the AWS IoT Device SDK for JavaScript
- MQTT publish/subscribe React web application using AWS Amplify, the AWS IoT Device SDK for JavaScript, and a Lambda function

Amplify is a set of tools and services that helps you build web and mobile applications that integrate with AWS services. For more information about Amplify, see Amplify Framework Documentation,.

Both examples perform the following steps.

1. When a user signs up for an account, the application creates an Amazon Cognito user pool and identity.
2. When a user authenticates, the application creates and attaches a policy to the identity. This gives the user publish and subscribe permissions.
3. The user can use the application to publish and subscribe to MQTT topics.

The first example uses the `AttachPolicy` API operation directly inside the authentication operation. The following example demonstrates how to implement this API call inside a React web application that uses Amplify and the AWS IoT Device SDK for JavaScript.

```
function attachPolicy(id, policyName) {
    var Iot = new AWS.Iot({region: AWSConfiguration.region, apiVersion: 
 AWSConfiguration.apiVersion, endpoint: AWSConfiguration.endpoint});
    var params = {policyName: policyName, target: id};
```

```
    console.log("Attach IoT Policy: " + policyName + " with cognito identity id: " +
 id);
    Iot.attachPolicy(params, function(err, data) {
         if (err) {
               if (err.code !== 'ResourceAlreadyExistsException') {
                  console.log(err);
               }
          }
         else  {
            console.log("Successfully attached policy with the identity", data);
         }
     });
}
```

This code appears in the AuthDisplay.js file.

The second example implements the `AttachPolicy` API operation in a Lambda function. The following example shows how the Lambda uses this API call.

```
iot.attachPolicy(params, function(err, data) {
     if (err) {
           if (err.code !== 'ResourceAlreadyExistsException') {
              console.log(err);
              res.json({error: err, url: req.url, body: req.body});
           }
      }
     else  {
        console.log(data);
        res.json({success: 'Create and attach policy call succeed!', url: req.url,
 body: req.body});
     }
 });
```

This code appears inside the `iot.GetPolicy` function in the app.js file.

> **Note**
>
> When you call the function with AWS credentials that you obtain through Amazon Cognito Identity pools, the context object in your Lambda function contains a value for `context.cognito_identity_id`. For more information, see the following.
>
> - AWS Lambda context object in Node.js
> - AWS Lambda context object in Python
> - AWS Lambda context object in Ruby
> - AWS Lambda context object in Java
> - AWS Lambda context object in Go
> - AWS Lambda context object in C#
> - AWS Lambda context object in PowerShell

Authorizing direct calls to AWS services using AWS IoT Core credential provider

Devices can use X.509 certificates to connect to AWS IoT Core using TLS mutual authentication protocols. Other AWS services do not support certificate-based authentication, but they can be called using AWS credentials in AWS Signature Version 4 format. The Signature Version 4 algorithm normally requires the caller to have an access key ID and a secret access key. AWS IoT Core has a credentials provider that allows you to use the built-in X.509 certificate as the unique device identity to authenticate AWS requests. This eliminates the need to store an access key ID and a secret access key on your device.

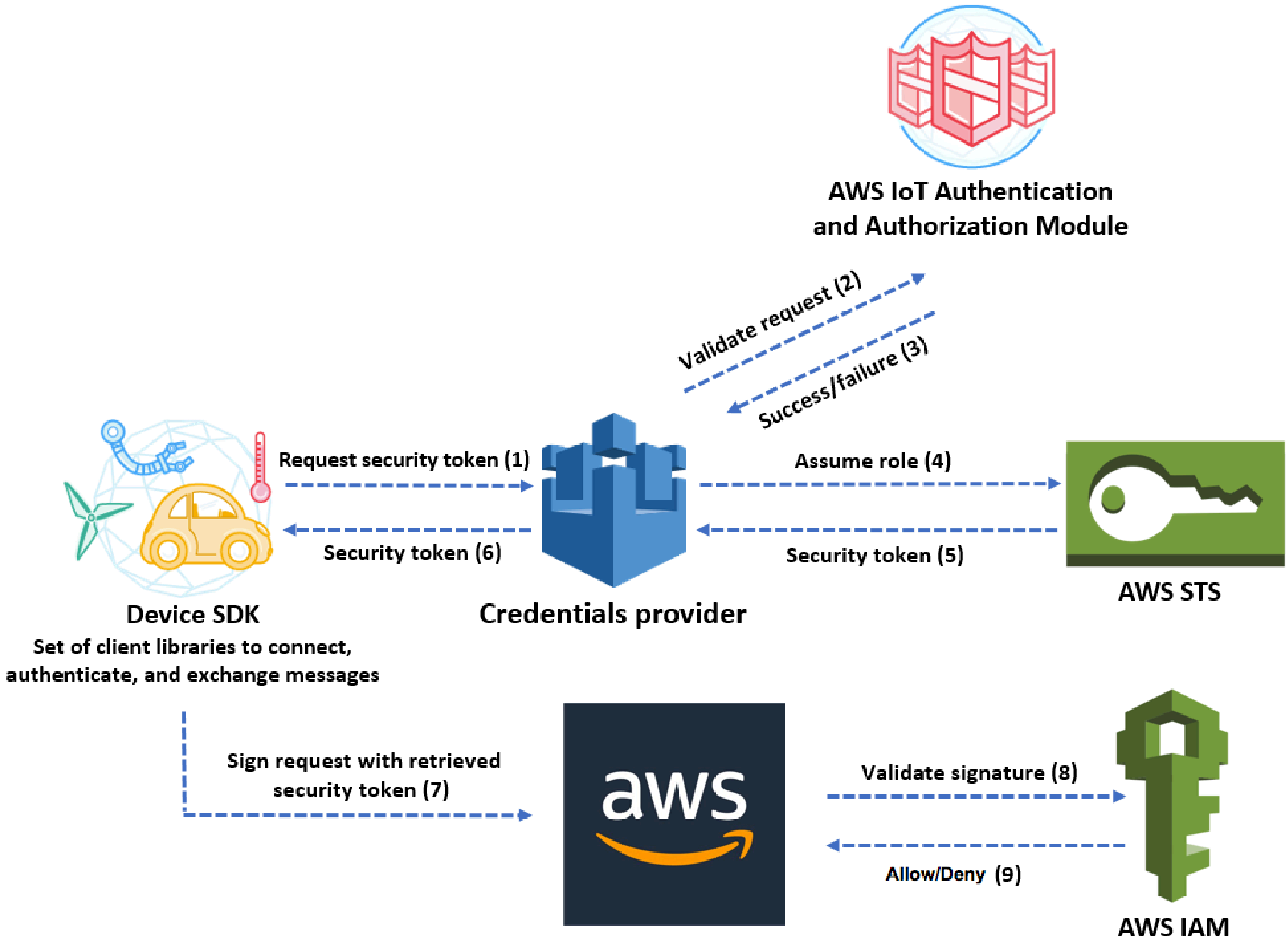
The credentials provider authenticates a caller using an X.509 certificate and issues a temporary, limited-privilege security token. The token can be used to sign and authenticate any AWS request. This way of authenticating your AWS requests requires you to create and configure an AWS Identity and Access Management (IAM) role and attach appropriate IAM policies to the role so that the credentials provider can assume the role on your behalf. For more information about AWS IoT Core and IAM, see Identity and access management for AWS IoT.

AWS IoT requires devices to send the Server Name Indication (SNI) extension to the Transport Layer Security (TLS) protocol and provide the complete endpoint address in the `host_name` field. The `host_name` field must contain the endpoint you are calling, and it must be:

- The `endpointAddress` returned by `aws iot` describe-endpoint `--endpoint-type iot:CredentialProvider`.

Connections attempted by devices without the correct `host_name` value will fail.

The following diagram illustrates the credentials provider workflow.



1. The AWS IoT Core device makes an HTTPS request to the credentials provider for a security token. The request includes the device X.509 certificate for authentication.
2. The credentials provider forwards the request to the AWS IoT Core authentication and authorization module to validate the certificate and verify that the device has permission to request the security token.

3. If the certificate is valid and has permission to request a security token, the AWS IoT Core authentication and authorization module returns success. Otherwise, it sends an exception to the device.
4. After successfully validating the certificate, the credentials provider invokes the AWS Security Token Service (AWS STS) to assume the IAM role that you created for it.
5. AWS STS returns a temporary, limited-privilege security token to the credentials provider.
6. The credentials provider returns the security token to the device.
7. The device uses the security token to sign an AWS request with AWS Signature Version 4.
8. The requested service invokes IAM to validate the signature and authorize the request against access policies attached to the IAM role that you created for the credentials provider.
9. If IAM validates the signature successfully and authorizes the request, the request is successful. Otherwise, IAM sends an exception.

The following section describes how to use a certificate to get a security token. It is written with the assumption that you have already registered a device and created and activated your own certificate for it.

How to use a certificate to get a security token

1. Configure the IAM role that the credentials provider assumes on behalf of your device. Attach the following trust policy to the role.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Principal": {"Service": "credentials.iot.amazonaws.com"},
        "Action": "sts:AssumeRole"
    }
}
```

   For each AWS service that you want to call, attach an access policy to the role. The credentials provider supports the following policy variables:

   - `credentials-iot:ThingName`
   - `credentials-iot:ThingTypeName`

- `credentials-iot:AwsCertificateId`

When the device provides the thing name in its request to an AWS service, the credentials provider adds `credentials-iot:ThingName` and `credentials-iot:ThingTypeName` as context variables to the security token. The credentials provider provides `credentials-iot:AwsCertificateId` as a context variable even if the device doesn't provide the thing name in the request. You pass the thing name as the value of the `x-amzn-iot-thingname` HTTP request header.

These three variables work for IAM policies only, not AWS IoT Core policies.

2. Make sure that the user who performs the next step (creating a role alias) has permission to pass the newly created role to AWS IoT Core. The following policy gives both `iam:GetRole` and `iam:PassRole` permissions to an AWS user. The `iam:GetRole` permission allows the user to get information about the role that you've just created. The `iam:PassRole` permission allows the user to pass the role to another AWS service.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "iam:GetRole",
            "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::123456789012:role/your role name"
    }
}
```

3. Create an AWS IoT Core role alias. The device that is going to make direct calls to AWS services must know which role ARN to use when connecting to AWS IoT Core. Hard-coding the role ARN is not a good solution because it requires you to update the device whenever the role ARN changes. A better solution is to use the `CreateRoleAlias` API to create a role alias that points to the role ARN. If the role ARN changes, you simply update the role alias. No change is required on the device. This API takes the following parameters:

`roleAlias`

Required. An arbitrary string that identifies the role alias. It serves as the primary key in the role alias data model. It contains 1-128 characters and must include only alphanumeric characters and the =, @, and - symbols. Uppercase and lowercase alphabetic characters are allowed. Role alias names are case sensitive.

`roleArn`

Required. The ARN of the role to which the role alias refers.

`credentialDurationSeconds`

Optional. How long (in seconds) the credential is valid. The minimum value is 900 seconds (15 minutes). The maximum value is 43,200 seconds (12 hours). The default value is 3,600 seconds (1 hour).

> **Important**
>
> The AWS IoT Core Credential Provider can issue a credential with a maximum lifetime is 43,200 seconds (12 hours). Having the credential be valid for up to 12 hours can help reduce the number of calls to the credential provider by caching the credential longer.
> The `credentialDurationSeconds` value must be less than or equal to the maximum session duration of the IAM role that the role alias references. For more information, see Modifying a role maximum session duration (AWS API) from the AWS Identity and Access Management User Guide.

For more information about this API, see CreateRoleAlias.

4. Attach a policy to the device certificate. The policy attached to the device certificate must grant the device permission to assume the role. You do this by granting permission for the `iot:AssumeRoleWithCertificate` action to the role alias, as in the following example.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
                "Action": "iot:AssumeRoleWithCertificate",
                "Resource": "arn:aws:iot:us-east-1:123456789012:rolealias/your role
 alias"
            }
        ]
}
```

5. Make an HTTPS request to the credentials provider to get a security token. Supply the following information:

   - *Certificate*: Because this is an HTTP request over TLS mutual authentication, you must provide the certificate and the private key to your client while making the request. Use the same certificate and private key you used when you registered your certificate with AWS IoT Core.

     To make sure your device is communicating with AWS IoT Core (and not a service impersonating it), see [Server Authentication](), follow the links to download the appropriate CA certificates, and then copy them to your device.

   - *RoleAlias*: The name of the role alias that you created for the credentials provider. Role alias names are case sensitive and must match the role alias created in AWS IoT Core.

   - *ThingName*: The thing name that you created when you registered your AWS IoT Core thing. This is passed as the value of the `x-amzn-iot-thingname` HTTP header. This value is required only if you are using thing attributes as policy variables in AWS IoT Core or IAM policies.

   > **Note**
   >
   > The *ThingName* that you provide in `x-amzn-iot-thingname` must match the name of the AWS IoT Thing resource assigned to a cert. If it doesn't match, a 403 error is returned.

   Run the following command in the AWS CLI to obtain the credentials provider endpoint for your AWS account. For more information about this API, see [DescribeEndpoint](). For FIPS-enabled endpoints, see [AWS IoT Core - credential provider endpoints]().

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

The following JSON object is sample output of the **describe-endpoint** command. It contains the `endpointAddress` that you use to request a security token.

```
{
    "endpointAddress": "your_aws_account_specific_prefix.credentials.iot.your
 region.amazonaws.com"
}
```

Use the endpoint to make an HTTPS request to the credentials provider to return a security token. The following example command uses `curl`, but you can use any HTTP client.

Note

The *roleAlias* name is case sensitive and must match the role alias created in AWS IoT.

```
curl --cert your certificate --key your private key -H "x-amzn-iot-thingname: your
 thing name" --cacert AmazonRootCA1.pem https://your endpoint /role-aliases/your
 role alias/credentials
```

This command returns a security token object that contains an `accessKeyId`, a `secretAccessKey`, a `sessionToken`, and an expiration. The following JSON object is sample output of the `curl` command.

```
    {"credentials":{"accessKeyId":"access key","secretAccessKey":"secret access
 key","sessionToken":"session token","expiration":"2018-01-18T09:18:06Z"}}
```

You can then use the `accessKeyId`, `secretAccessKey`, and `sessionToken` values to sign requests to AWS services. For an end-to-end demonstration, see [How to Eliminate the Need for Hard-Coded AWS Credentials in Devices by Using the AWS IoT Credential Provider]() blog post on the *AWS Security Blog*.

Cross account access with IAM

AWS IoT Core allows you to enable a principal to publish or subscribe to a topic that is defined in an AWS account not owned by the principal. You configure cross account access by creating an IAM policy and IAM role and then attaching the policy to the role.

First, create a customer managed IAM policy as described in [Creating IAM Policies](), just like you would for other users and certificates in your AWS account.

For devices registered in AWS IoT Core registry, the following policy grants permission to devices connect to AWS IoT Core using a client ID that matches the device's thing name and to publish to the `my/topic/thing-name` where *thing-name* is the device's thing name:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/
${iot:Connection.Thing.ThingName}"
            ]
        }
    ]
}
```

For devices not registered in AWS IoT Core registry, the following policy grants permission to a device to use the thing name `client1` registered in your account's (123456789012) AWS IoT Core registry to connect to AWS IoT Core and to publish to a client ID-specific topic whose name is prefixed with `my/topic/`:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/client1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/my/topic/${iot:ClientId}"
            ]
        }
    ]
}
```

Next, follow the steps in [Creating a role to delegate permissions to an IAM user](). Enter the account ID of the AWS account with which you want to share access. Then, in the final step, attach the policy you just created to the role. If, at a later time, you need to modify the AWS account ID to which you are granting access, you can use the following trust policy format to do so:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
                "Principal": {
                    "AWS": "arn:aws:iam::567890123456:user/MyUser"
            },
                "Action": "sts:AssumeRole"
            }
        ]
}
```

Data protection in AWS IoT Core

The AWS shared responsibility model applies to data protection in AWS IoT Core. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with AWS IoT or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

For more information about data protection, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

AWS IoT devices gather data, perform some manipulation on that data, and then send that data to another web service. You might choose to store some data on your device for a short period of time. You're responsible for providing any data protection on that data at rest. When your device sends data to AWS IoT, it does so over a TLS connection as discussed later in this section. AWS IoT devices can send data to any AWS service. For more information about each service's data security, see the documentation for that service. AWS IoT can be configured to write logs to CloudWatch Logs and log AWS IoT API calls to AWS CloudTrail. For more information about data security for these services, see Authentication and Access Control for Amazon CloudWatch and Encrypting CloudTrail Log Files with AWS KMS managed keys.

Data encryption in AWS IoT

By default, all AWS IoT data in transit and at rest is encrypted. Data in transit is encrypted using TLS, and data at rest is encrypted using AWS owned keys. AWS IoT supports customer managed AWS KMS keys (KMS keys) from AWS Key Management Service (AWS KMS). However, Device Advisor and AWS IoT Wireless use only an AWS owned key to encrypt customer data.

Transport security in AWS IoT Core

TLS (Transport Layer Security) is a cryptographic protocol that is designed for secure communication over a computer network. The AWS IoT Core Device Gateway requires customers to encrypt all communication while in-transit by using TLS for connections from devices to the Gateway. TLS is used to achieve confidentiality of the application protocols (MQTT, HTTP, and WebSocket) supported by AWS IoT Core. TLS support is available in a number of programming languages and operating systems. Data within AWS is encrypted by the specific AWS service. For more information about data encryption on other AWS services, see the security documentation for that service.

Contents

TLS protocols

AWS IoT Core supports the following versions of the TLS protocol:

- TLS 1.3
- TLS 1.2

With AWS IoT Core, you can configure the TLS settings (for TLS 1.2 and TLS 1.3) in domain configurations. For more information, see ???.

Security policies

A security policy is a combination of TLS protocols and their ciphers that determine which protocols and ciphers are supported during TLS negotiations between a client and a server. You can configure your devices to use predefined security policies based on your needs. Note that AWS IoT Core doesn't support custom security policies.

You can choose one of the predefined security policies for your devices when connecting them to AWS IoT Core. The names of the most recent predefined security policies in AWS IoT Core include version information based on the year and month that they were released. The default predefined security policy is `IoTSecurityPolicy_TLS13_1_2_2022_10`. To specify a security policy, you can use the AWS IoT console or the AWS CLI. For more information, see ???.

The following table describes the most recent predefined security policies that AWS IoT Core supports. The `IotSecurityPolicy_` has been removed from policy names in the heading row so that they fit.

Security policy	TLS13_1_3_2022_10	TLS13_1_2_2022_10	TLS12_1_2_2022_10	TLS12_1_0_2016_01*		TLS12_1_0_2015_01*	
TCP Port	443/8443/8883	443/8443/8883	443/8443/8883	443	8443/8883	443	8443/8883
TLS Protocols							
TLS 1.2		✓	✓	✓	✓	✓	✓
TLS 1.3	✓	✓					
TLS Ciphers							
TLS_AES_128_GCM_SHA256	✓	✓					
TLS_AES_256_GCM_SHA384	✓	✓					
TLS_CHACHA20_POLY1305_SHA256	✓	✓					
ECDHE-RSA-AES128-GCM-SHA256		✓	✓	✓	✓	✓	✓
ECDHE-RSA-AES128-SHA256		✓	✓	✓	✓	✓	✓

Security policy	TLS13_1_3_2022_10	TLS13_1_2_2022_10	TLS12_1_2_2022_10	TLS12_1_0_2016_01*		TLS12_1_0_2015_01*	
ECDHE-RSA-AES128-SHA		✓	✓	✓	✓	✓	✓
ECDHE-RSA-AES256-GCM-SHA384		✓	✓	✓	✓	✓	✓
ECDHE-RSA-AES256-SHA384		✓	✓	✓	✓	✓	✓
ECDHE-RSA-AES256-SHA		✓	✓	✓	✓	✓	✓
AES128-GCM-SHA256		✓	✓	✓	✓	✓	✓
AES128-SHA256		✓	✓	✓		✓	✓
AES128-SHA		✓	✓	✓	✓	✓	✓
AES256-GCM-SHA384		✓	✓	✓	✓	✓	✓

Security policy	TLS13_1_3_2022_10	TLS13_1_2_2022_10	TLS12_1_2_2022_10	TLS12_1_0_2016_01*		TLS12_1_0_2015_01*	
AES256-SHA256		✓	✓	✓	✓	✓	✓
AES256-SHA		✓	✓	✓	✓	✓	✓
DHE-RSA-AES256-SHA						✓	✓
ECDHE-ECDSA-AES128-GCM-SHA256		✓	✓	✓	✓	✓	✓
ECDHE-ECDSA-AES128-SHA256		✓	✓	✓	✓	✓	✓
ECDHE-ECDSA-AES128-SHA		✓	✓	✓	✓	✓	✓
ECDHE-ECDSA-AES256-GCM-SHA384		✓	✓	✓	✓	✓	✓

Security policy	TLS13_1_3 _2022_10	TLS13_1_2 _2022_10	TLS12_1_2 _2022_10	TLS12_1_0 _2016_01*		TLS12_1_0 _2015_01*	
ECDHE-ECDSA-AES256-SHA384		✓	✓	✓	✓	✓	✓
ECDHE-ECDSA-AES256-SHA		✓	✓	✓	✓	✓	✓

Note

`TLS12_1_0_2016_01` is only available in the following AWS Regions: ap-east-1, ap-northeast-2, ap-south-1, ap-southeast-2, ca-central-1, cn-north-1, cn-northwest-1, eu-north-1, eu-west-2, eu-west-3, me-south-1, sa-east-1, us-east-2, us-gov-west-1, us-gov-west-2, us-west-1.

`TLS12_1_0_2015_01` is only available in the following AWS Regions: ap-northeast-1, ap-southeast-1, eu-central-1, eu-west-1, us-east-1, us-west-2.

Important notes for transport security in AWS IoT Core

For devices that connect to AWS IoT Core using MQTT, TLS encrypts the connection between the devices and the broker, and AWS IoT Core uses TLS client authentication to identify devices. For more information, see Client authentication. For devices that connect to AWS IoT Core using HTTP, TLS encrypts the connection between the devices and the broker, and authentication is delegated to AWS Signature Version 4. For more information, see Signing requests with Signature Version 4 in the *AWS General Reference*.

When you connect devices to AWS IoT Core, sending the Server Name Indication (SNI) extension is not required but highly recommended. To use features such as multi-account registration, custom domains, VPC endpoints, and configured TLS policies, you must use the SNI extension and provide the complete endpoint address in the `host_name` field. The `host_name` field must contain the endpoint you are calling. That endpoint must be one of the following:

- The `endpointAddress` returned by `aws iot describe-endpoint --endpoint-type iot:Data-ATS`
- The `domainName` returned by `aws iot describe-domain-configuration --domain-configuration-name "domain_configuration_name"`

Connections attempted by devices with the incorrect or invalid `host_name` value will fail. AWS IoT Core will log failures to CloudWatch for the authentication type of Custom Authentication.

AWS IoT Core doesn't support the SessionTicket TLS extension.

Transport security for LoRaWAN wireless devices

LoRaWAN devices follow the security practices described in LoRaWAN ™ SECURITY: A White Paper Prepared for the LoRa Alliance™ by Gemalto, Actility, and Semtech.

For more information about transport security with LoRaWAN devices, see LoRaWAN data and transport security.

Data encryption in AWS IoT

Data protection refers to protecting data while in-transit (as it travels to and from AWS IoT Core) and at rest (while it is stored on devices or by other AWS services). All data sent to AWS IoT Core is sent over an TLS connection using MQTT, HTTPS, and WebSocket protocols, making it secure by default while in transit. AWS IoT Core collects data from devices and then sends it to other AWS services for further processing. For more information about data encryption on other AWS services, see the security documentation for that service. For more information, see Data encryption at rest.

FreeRTOS provides a PKCS#11 library that abstracts key storage, accessing cryptographic objects and managing sessions. It is your responsibility to use this library to encrypt data at rest on your devices. For more information, see FreeRTOS Public Key Cryptography Standard (PKCS) #11 Library.

Data encryption at rest in AWS IoT Core

By default, all AWS IoT Core data at rest is encrypted using AWS owned keys. AWS IoT Core also supports symmetric customer managed keys from AWS Key Management Service (AWS KMS). With customer managed keys, you can create, own, and manage the AWS KMS keys in your AWS account. AWS IoT Core will use your KMS keys to encrypt your data at rest. You have full control over these KMS keys, including creating and maintaining their key policies. You can also configure IAM policies for the roles that access AWS KMS to control permissions for these keys.

AWS owned keys

AWS owned keys are a collection of KMS keys that an AWS service owns and manages for use in multiple AWS accounts. AWS services can use AWS owned keys to protect your data. By default, AWS IoT Core encrypts data at rest using AWS owned keys. These keys are managed by the service. You can't view, manage, or use AWS owned keys. However, you don't need to take any actions to protect these keys.

For more information about AWS owned keys, see AWS owned keys in the *AWS Key Management Service Developer Guide*.

Customer managed keys

Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these AWS KMS keys, including creating and maintaining their key policies. You can also configure IAM policies for the roles that access AWS KMS to control permissions for these keys. You can configure AWS IoT Core to use customer managed KMS keys to encrypt your data.

For more information about customer managed keys, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

To opt in to customer managed keys in AWS IoT Core, follow these steps:

Topics

Step 1: Create a customer managed key

You can create a symmetric customer managed key by using the AWS KMS console or the AWS KMS CLI commands. The `keySpec` must be `SYMMETRIC_DEFAULT` and the `keyUsage` must be `ENCRYPT_DECRYPT`.

> **Note**
>
> AWS IoT Core only supports AWS KMS keys with `SYMMETRIC_DEFAULT` key spec and `ENCRYPT_DECRYPT` key usage for customer managed keys.

The following is an example AWS CLI command to create a KMS key which can be used with AWS IoT Core for customer managed keys.

```
aws kms create-key --key-spec SYMMETRIC_DEFAULT --key-usage ENCRYPT_DECRYPT --region
 us-west-2
```

The following is an example output of the command.


```
{
    "KeyMetadata": {
        "AWSAccountId": "111122223333",
        "KeyId": "1234abcd-12ab-34cd-56ef-1234567890ab",
        "Arn": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        "CreationDate": "2024-09-19T11:45:23.982000-07:00",
        "Enabled": true,
        "Description": "",
        "KeyUsage": "ENCRYPT_DECRYPT",
        "KeyState": "Enabled",
        "Origin": "AWS_KMS",
        "KeyManager": "CUSTOMER",
        "CustomerMasterKeySpec": "SYMMETRIC_DEFAULT",
        "KeySpec": "SYMMETRIC_DEFAULT",
        "EncryptionAlgorithms": [
            "SYMMETRIC_DEFAULT"
        ],
        "MultiRegion": false
    }
}
```

For more information, see Creating a symmetric customer managed key in the *AWS Key Management Service Developer Guide*.

Key policy

When creating a customer managed key, you can specify a key policy. Key policies control access to your customer managed key. Every customer managed key must have exactly one key policy, which contains statements that determine who can use the key and how they can use it. For more information, see Key policies in the *AWS Key Management Service Developer Guide*.

AWS IoT Core uses an IAM role in your account to access your customer managed key. If you're using a custom key policy, make sure the IAM role created on this key has the following permissions:

- `kms:DescribeKey`
- `kms:Decrypt`
- `kms:Encrypt`
- `kms:GenerateDataKeyWithoutPlaintext`
- `kms:ReEncryptTo`
- `kms:ReEncryptFrom`

Step 2: Create an IAM role to grant AWS IoT Core permissions to use the KMS key

For AWS IoT Core to use the KMS key you created to encrypt your data at rest, you also need to create an IAM role in your account, which AWS IoT Core can assume to access the KMS key.

The role must have the following trust policy to allow AWS IoT Core to assume the role.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Principal": {
            "Service": "iot.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "111122223333"
```

```
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:iot:us-west-2:111122223333:*"
            }
        }
    }
}
```

Ensure the IAM policies attached to the IAM role have the following permissions on the KMS key:

- `kms:DescribeKey`
- `kms:Decrypt`
- `kms:Encrypt`
- `kms:GenerateDataKeyWithoutPlaintext`
- `kms:ReEncryptTo`
- `kms:ReEncryptFrom`

The following is an example IAM policy with the required permissions for customer managed keys.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowIoTToAccessKMSResource",
            "Effect": "Allow",
            "Action": [
                "kms:DescribeKey",
                "kms:Decrypt",
                "kms:Encrypt",
                "kms:ReEncryptTo",
                "kms:ReEncryptFrom",
                "kms:GenerateDataKeyWithoutPlaintext"
            ],
            "Resource": [
                "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
            ],
            "Condition": {
                "StringEquals": {
                    "kms:EncryptionContext:aws-crypto-ec:vendor": "iot.amazonaws.com"
```

```
                }
            }
        }
    ]
}
```

For more information, see [Create a role to delegate permissions to an IAM user] in the *AWS Identity and Access Management User Guide*.

Step 3: Opt in to customer managed keys in AWS IoT Core

After you complete all the previous steps, run the `update-encryption-configuration` CLI command to opt in using customer managed keys in AWS IoT Core. When you opt in to customer managed keys, all AWS IoT Core resources in your AWS account will be encrypted using the specified AWS KMS key.

1. To opt in to customer managed keys in AWS IoT Core using AWS CLI, run the `update-encryption-configuration` CLI command.

```
aws iot update-encryption-configuration --encryption-type "CUSTOMER_MANAGED_KMS_KEY"
 \
--kms-access-role-arn "arn:aws:iam::111122223333:role/myrole" \
--kms-key-arn "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab" --region us-west-2
```

2. To verify customer managed keys in AWS IoT Core using AWS CLI, run the `describe-encryption-configuration` CLI command:

```
aws iot describe-encryption-configuration --region us-west-2
```

If you have enabled customer managed keys in AWS IoT Core, the output can look like the following:

```
{
    "encryptionType": "CUSTOMER_MANAGED_KMS_KEY",
    "kmsKeyArn": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    "kmsAccessRoleArn": "arn:aws:iam::111122223333:role/myrole",
    "configurationDetails": {
        "configurationStatus": "HEALTHY"
    },
```

```
    "lastModifiedDate": "2024-09-26T22:01:02.365000-07:00"
}
```

The `lastModifiedDate` field indicates the date when the encryption configuration was last updated.

If you haven't enabled customer managed keys, the output can look like the following:

```
{
    "encryptionType": "AWS_OWNED_KMS_KEY",
    "lastModifiedDate": "2024-09-26T22:01:02.365000-07:00"
}
```

Step 4: Additional permissions required for AWS IoT Core control plane operations

After you opt in to customer managed keys, all the AWS IoT Core resources belonging to your AWS account are encrypted with the provided KMS key. All control plane operations now require the caller to have `kms:Decrypt` permissions on the KMS key in addition to the permissions required for the specific operation on the AWS IoT Core resource. If the caller does not have `kms:Decrypt` permission and they make an API call that requires encryption or decryption of data (for example, `GetPolicy`), they will receive an `UnauthorizedException`.

For example, when you call `GetPolicy`, you need both `iot:GetPolicy` and `kms:Decrypt` permissions on your customer managed KMS key for the API call to succeed.

> **Note**
>
> When updating IAM users or roles to grant AWS KMS permissions on the key used for your encryption configuration, ensure the KMS key policy also grants the required permissions to the respective IAM users or roles.

AWS KMS permissions for `UpdateEncryptionConfiguration`

The `UpdateEncryptionConfiguration` API call needs the following AWS KMS permissions on the KMS key to be able to opt in to customer managed keys or to modify the key configuration:

- `kms:DescribeKey`
- `kms:Decrypt`

- `kms:Encrypt`
- `kms:GenerateDataKeyWithoutPlaintext`
- `kms:ReEncryptTo`
- `kms:ReEncryptFrom`

AWS KMS permissions for all other control plane APIs

Most control plane APIs require `kms:Decrypt` permissions when customer managed keys are enabled. However, certain APIs do not require these additional permissions:

APIs that do not require AWS KMS permissions

The `List*` and `Delete*` APIs do not fall into this bucket. Customers can always invoke any `List*` or `Delete*` control plane API and those API calls would succeed even if the caller does not have `kms:Decrypt` permission. These API calls will succeed even if your customer managed key is unhealthy as `List*` and `Delete*` APIs do not do any decryption.

- **List* APIs** – All listing operations (for example, `ListThings`, `ListPolicies`, `ListCertificates`)
- **Delete* APIs** – All deletion operations (for example, `DeleteThing`, `DeletePolicy`, `DeleteCertificate`)

Step 5: Managing keys

AWS IoT Core runs periodic checks on your customer managed key configuration to ensure encrypt and decrypt operations are not impacted. These health checks run once every minute and verify AWS IoT Core's ability to access and use both the AWS KMS key and the associated IAM role for encrypt and decrypt operations.

HEALTHY

AWS IoT Core can successfully access the AWS KMS key through the specified IAM role and perform encryption/decryption operations. All components function correctly.

UNHEALTHY

AWS IoT Core can't access or use the AWS KMS key. This prevents new encryption operations and may impact service functionality. The `errorCode` field indicates whether the issue is with the key or the IAM role.

Customer actions that can impact key health

Several customer actions can cause the key health status to change from `HEALTHY` to `UNHEALTHY`:

Key-related actions

- **Deleting an AWS KMS key** – When you schedule a key deletion, it's in a `Pending deletion` status and can't be used
- **Disabling an AWS KMS key** – When you disable a KMS key, it can no longer be used for encrypt / decrypt operations
- **Scheduling key for deletion** – Key becomes unusable when deletion completes
- **Modifying key policy** – Removing necessary permissions for AWS IoT Core access
- **Changing key usage permissions** – Restricting required AWS KMS actions

IAM role-related actions

- **Deleting the IAM role** – AWS IoT Core can't assume the role to access the key
- **Modifying role permissions** – Removing required AWS KMS permissions from the role policy
- **Changing trust policy** – Preventing AWS IoT Core service from assuming the role
- **Adding restrictive conditions** – Conditions that prevent AWS IoT Core from using the role

Account-level actions

- **Cross-account key access changes** – Modifying permissions for keys in different accounts
- **Service Control Policies (SCPs)** – Organization-level policies that restrict AWS KMS access
- **Account-level IAM policies** – Policies that override or conflict with key access

> **Important**
>
> Any changes to AWS KMS keys, IAM roles, or policies used by AWS IoT Core should be tested in development environments first. Monitor the key health status closely after making any changes to ensure AWS IoT Core functionality is not impacted.

Updating encryption configuration

Update your encryption configuration in AWS IoT Core to change from one customer managed key to another, or between AWS owned keys and customer managed keys.

To change the configuration to a different customer managed key:

1. Create a new customer managed key following the steps in [the section called "Step 1: Create a customer managed key"]().
2. Update your IAM role policy to include permissions for both the old and new keys during the update period.
3. Update your encryption configuration to use the new key:

```
aws iot update-encryption-configuration --encryption-type "CUSTOMER_MANAGED_KMS_KEY"
 \
--kms-access-role-arn "arn:aws:iam::111122223333:role/myrole" \
--kms-key-arn "arn:aws:kms:us-west-2:111122223333:key/new-key-id"
```

To change the configuration from customer managed keys back to AWS owned keys:

```
aws iot update-encryption-configuration --encryption-type "AWS_OWNED_KMS_KEY"
```

> **Note**
>
> When updating the encryption configuration for new customer managed keys, ensure both the old and new keys remain accessible for the operation to succeed.

Common failure scenarios and impacts

The following table describes common failure scenarios when keys are deleted or deactivated:

Scenario	Immediate Impact	Long-term Consequences
Key disabled	All new encryption/decryption operations fail immediately	Service disruption until the key is re-enabled or replaced
Key scheduled for deletion	Key status is changed to pending deletion and all encryption/decryption operations will fail	Automatic service failure when deletion completes
Key permanently deleted	Immediate and permanent failure of all operations	Permanent data loss and inability to recover encrypted data

Scenario	Immediate Impact	Long-term Consequences
Key policy modified incorrectly	AWS IoT Core loses access permissions to the key	Service failures until policy is corrected
IAM role deleted	AWS IoT Core can't assume role to access key	Complete encryption service failure
IAM role is modified incorrectly	AWS IoT Core can't assume role or use role to access key	Service failures until IAM role is corrected

Prevention and best practices

To prevent accidental key deletion or deactivation and minimize the risk of service failures:

Implement key lifecycle policies

Establish clear procedures for key creation, rotation, and retirement. Document which keys are used by which AWS IoT Core resources and maintain an inventory of active keys.

Use IAM policies to restrict key deletion

Create IAM policies that prevent unauthorized users from deleting or disabling critical encryption keys. Use conditions to require additional approval for key deletion operations.

Enable CloudTrail logging

Monitor all AWS KMS key operations through CloudTrail to detect unauthorized or accidental key management activities. Set up alerts for key deletion, disabling, or policy changes.

Test key replacement procedures

Regularly test your key replacement procedures in non-production environments to ensure you can quickly recover from key-related failures.

Maintain key backups

While you can't export AWS KMS key material, maintain detailed records of key ARNs, policies, and associated AWS IoT Core configurations to facilitate rapid key replacement if needed.

Monitor key health

Continuously monitor the `CMK.Health` metric and set up automated alerts for key health status changes. Implement automated responses to quickly address key-related issues.

> **Important**
>
> Always test key update procedures in development environments before implementing them in production. Have a documented rollback plan and ensure that key replacement procedures can be executed quickly in case of emergencies.

Step 6: Monitoring key health

As part of the periodic checks AWS IoT Core runs, CloudWatch metrics and logs are emitted to provide visibility on the health status of your customer managed key configuration

AWS IoT Core emits the `CMK.Health` metric to CloudWatch at least once every minute. The metric provides information about the health status of the customer managed keys used by AWS IoT Core for encrypting and decrypting your data.

The `CMK.Health` metric can have the following values:

- The value is `1`: AWS IoT Core is able to use the encryption keys successfully for encrypting and decrypting your data.
- The value is `0`: AWS IoT Core is unable to use the encryption keys for encrypting and decrypting your data.

AWS IoT Core also emits AWS IoT V2 logs when the health status of the encryption keys changes. These logs provide additional details about the health status update. To view these logs, you must enable AWS IoT V2 logs. The `HEALTHY` logs are emitted at `INFO` level, and the `UNHEALTHY` logs are emitted at `ERROR` level. For more information about the log levels, see Log levels.

The following examples are CloudWatch log entries emitted by AWS IoT Core to indicate the health status update of the customer managed keys.

To effectively monitor and respond to key health status changes:

1. **Set up CloudWatch alarms** for the `CMK.Health` metric:

```
aws cloudwatch put-metric-alarm --region us-west-2 \
  --alarm-name "IoTCore-CMK-Health-Alert" \
  --alarm-description "Alert when IoT Core CMK health is unhealthy" \
  --metric-name "CMK.Health" \
  --namespace "AWS/IoT" \
  --statistic "Minimum" \
  --period 300 \
  --evaluation-periods 1 \
  --threshold 1 \
  --comparison-operator "LessThanThreshold" \
  --alarm-actions "arn:aws:sns:us-west-2:111122223333:iot-alerts"
```

2. **Enable AWS IoT V2 logging** to capture detailed health status change events with error codes and messages.
3. **Check configuration status** for troubleshooting:

```
aws iot describe-encryption-configuration --region us-west-2
```

4. **Investigate UNHEALTHY status** by examining the `errorCode` field:
   - `KMS_KEY_VALIDATION_ERROR` – Issue with the AWS KMS key (disabled, deleted, or policy problems)
   - `ROLE_VALIDATION_ERROR` – Issue with the IAM role (deleted, policy problems, or trust issues)

From UNHEALTHY to HEALTHY

When the status of the encryption keys is updated from `UNHEALTHY` to `HEALTHY`, AWS IoT Core will emit an AWS IoT V2 log message in the following format.

```
{
    "timestamp": "2017-08-10 15:37:23.476",
    "logLevel": "INFO",
    "traceId": "8421693b-f4f0-4e4a-9235-0cff8bab897d",
    "accountId": "111122223333",
    "status": "SUCCESS",
    "cmkStatus": "HEALTHY",
    "kmsKeyArn": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    "kmsAccessRoleArn": "arn:aws:iam::111122223333:role/myrole",
    "eventType": "CmkHealthCheck"
```

```
}
```

From HEALTHY to UNHEALTHY

When the status of the encryption keys is updated from HEALTHY to UNHEALTHY, AWS IoT Core will emit an AWS IoT V2 log message in the following format.


```
{
    "timestamp": "2017-08-10 15:37:23.476",
    "logLevel": "ERROR",
    "traceId": "8421693b-f4f0-4e4a-9235-0cff8bab897d",
    "accountId": "111122223333",
    "status": "FAILURE",
    "cmkStatus": "UNHEALTHY",
    "errorCode": "KMS_KEY_VALIDATION_ERROR / ROLE_VALIDATION_ERROR",
    "errorMessage": "Error message on why there was a failure",
    "kmsKeyArn": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    "kmsAccessRoleArn": "arn:aws:iam::111122223333:role/myrole",
    "eventType": "CmkHealthCheck"
}
```


Warning

When key health becomes UNHEALTHY, AWS IoT Core operations fail immediately. If this occurs, review your key configurations, IAM role permissions, and policies. Monitor the CMK.Health metric for status changes. If operations continue to fail after reviewing your configurations, contact your account manager or the AWS Support Center for additional assistance.

AWS CloudTrail events

You can also monitor AWS IoT Core's usage of the KMS key for encrypt decrypt operations. AWS IoT Core will make DescribeKey, Decrypt, ReEncrypt, and GenerateDataKeyWithoutPlaintext operations on your KMS key to encrypt / decrypt data belonging to your AWS account stored at rest.

There are CloudTrail events for DescribeKey, Decrypt, ReEncrypt, and GenerateDataKeyWithoutPlaintext. These events monitor AWS KMS operations called by AWS IoT Core to access data encrypted by your customer managed key.

`Decrypt` example


```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "********************",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
                "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
                "accountId": "111122223333",
                "userName": "*****"
            },
            "attributes": {
                "creationDate": "2024-09-16T20:23:39Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "iot.amazonaws.com"
    },
    "eventTime": "2024-09-16T20:32:48Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "iot.amazonaws.com",
    "userAgent": "iot.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "kms-arn": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "aws-crypto-ec:vendor": "iot.amazonaws.com",
            "branch-key-id": "111122223333",
            "type": "branch:ACTIVE"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT",
        "keyId": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    "responseElements": null,
```

```
    "requestID": "1afb6d98-8388-455d-8b48-e62c9e0cf7f4",
    "eventID": "b59a5f16-0d98-46d8-a590-0e040a48b39b",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```


Identity and access management for AWS IoT

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS IoT resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting AWS IoT identity and access)
- **Service administrator** - determine user access and submit permission requests (see How AWS IoT works with IAM)
- **IAM administrator** - write policies to manage access (see AWS IoT identity-based policy examples)

Authenticating with IAM identities

In AWS IoT identities can be device (X.509) certificates, Amazon Cognito identities, or IAM users or groups. This topic discusses IAM identities only. For more information about the other identities that AWS IoT supports, see Client authentication.

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How AWS IoT works with IAM

Before you use IAM to manage access to AWS IoT, you should understand which IAM features are available to use with AWS IoT. To get a high-level view of how AWS IoT and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- AWS IoT identity-based policies
- AWS IoT resource-based policies
- Authorization based on AWS IoT tags
- AWS IoT IAM roles

AWS IoT identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. AWS IoT supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

The following table lists the IAM IoT actions, the associated AWS IoT API, and the resource the action manipulates.

Policy actions	AWS IoT API	Resources
iot:AcceptCertificateTransfer	AcceptCertificateTransfer	`arn:aws:iot: region:account-id :cert/cert-id` **Note** The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:AddThingToThingGroup	AddThingToThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` `arn:aws:iot: region:account-id :thing/thing-name`
iot:AssociateTargetsWithJob	AssociateTargetsWithJob	none

iot:AttachPolicy	AttachPolicy	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
		or
		`arn:aws:iot: region:account-id :cert/cert-id`

iot:AttachPrincipalPolicy	AttachPrincipalPolicy	`arn:aws:iot: region:account-id :cert/cert-id`
iot:AttachSecurityProfile	AttachSecurityProfile	`arn:aws:iot: region:account-id :securityprofile/ security-profile-name` `arn:aws:iot: region:account-id :dimension/ dimension-name`

Policy actions	AWS IoT API	Resources
iot:AttachThingPrincipal	AttachThingPrincipal	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:CancelCertificateTransfer	CancelCertificateTransfer	arn:aws:iot: *region*:*account-id* :cert/*cert-id* **Note** The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:CancelJob	CancelJob	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:CancelJobExecution	CancelJobExecution	arn:aws:iot: *region*:*account-id* :job/*job-id* arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:ClearDefaultAuthorizer	ClearDefaultAuthorizer	None
iot:CreateAuthorizer	CreateAuthorizer	arn:aws:iot: *region*:*account-id* :authorizer/ *authorizer-function-name*
iot:CreateCertificateFromCsr	CreateCertificateFromCsr	*
iot:CreateDimension	CreateDimension	arn:aws:iot: *region*:*account-id* :dimension/ *dimension-name*

Policy actions	AWS IoT API	Resources
iot:CreateJob	CreateJob	arn:aws:iot: *region*:*account-id* :job/*job-id* arn:aws:iot: *region*:*account-id* :thinggro up/ *thing-group-name* arn:aws:iot: *region*:*account-i d* :thing/*thing-name* arn:aws:iot: *region*:*account-id* :jobtempl ate/ *job-template-id*
iot:Creat eJobTemplate	CreateJob Template	arn:aws:iot: *region*:*account-id* :job/*job-id* arn:aws:iot: *region*:*account-id* :jobtempl ate/ *job-template-id*
iot:Creat eKeysAndC ertificate	CreateKey sAndCertificate	*
iot:CreatePolicy	CreatePolicy	arn:aws:iot: *region*:*account-i d* :policy/*policy-name*
iot:Creat ePolicyVersion	CreatePol icyVersion	arn:aws:iot: *region*:*account-i d* :policy/*policy-name* **Note** This must be an AWS IoT policy, not an IAM policy.
iot:Creat eRoleAlias	CreateRoleAlias	(parameter: roleAlias) arn:aws:iot: *region*:*account-id* :rolealia s/ *role-alias-name*

Policy actions	AWS IoT API	Resources

iot:CreateSecurityProfile	CreateSecurityProfile	`arn:aws:iot: region:account-id :securityprofile/ security-profile-name`
		
		`arn:aws:iot: region:account-id :dimension/ dimension-name`

iot:CreateThing	CreateThing	`arn:aws:iot: region:account-id :thing/thing-name`
iot:CreateThingGroup	CreateThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` for group being created and for parent group, if used
iot:CreateThingType	CreateThingType	`arn:aws:iot: region:account-id :thingtype/ thing-type-name`
iot:CreateTopicRule	CreateTopicRule	`arn:aws:iot: region:account-id :rule/rule-name`
iot:DeleteAuthorizer	DeleteAuthorizer	`arn:aws:iot: region:account-id :authorizer/ authorizer-name`
iot:DeleteCACertificate	DeleteCACertificate	`arn:aws:iot: region:account-id :cacert/cert-id`
iot:DeleteCertificate	DeleteCertificate	`arn:aws:iot: region:account-id :cert/cert-id`
iot:DeleteDimension	DeleteDimension	`arn:aws:iot: region:account-id :dimension/ dimension-name`
iot:DeleteJob	DeleteJob	`arn:aws:iot: region:account-id :job/job-id`
iot:DeleteJobTemplate	DeleteJobTemplate	`arn:aws:iot: region:account-id :job/job-template-id`

Policy actions	AWS IoT API	Resources
iot:DeleteJobExecution	DeleteJobExecution	arn:aws:iot: *region*:*account-id* :job/*job-id* arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:DeletePolicy	DeletePolicy	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:DeletePolicyVersion	DeletePolicyVersion	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:DeleteRegistrationCode	DeleteRegistrationCode	*
iot:DeleteRoleAlias	DeleteRoleAlias	arn:aws:iot: *region*:*account-id* :rolealias/ *role-alias-name*
iot:DeleteSecurityProfile	DeleteSecurityProfile	arn:aws:iot: *region*:*account-id* :securityprofile/ *security-profile-name* arn:aws:iot: *region*:*account-id* :dimension/ *dimension-name*
iot:DeleteThing	DeleteThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:DeleteThingGroup	DeleteThingGroup	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*
iot:DeleteThingType	DeleteThingType	arn:aws:iot: *region*:*account-id* :thingtype/ *thing-type-name*
iot:DeleteTopicRule	DeleteTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*

Policy actions	AWS IoT API	Resources
iot:DeleteV2LoggingLevel	DeleteV2LoggingLevel	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*
iot:DeprecateThingType	DeprecateThingType	arn:aws:iot: *region*:*account-id* :thingtype/ *thing-type-name*
iot:DescribeAuthorizer	DescribeAuthorizer	arn:aws:iot: *region*:*account-id* :authorizer/ *authorizer-function-name* (parameter: authorizerName) none
iot:DescribeCACertificate	DescribeCACertificate	arn:aws:iot: *region*:*account-id* :cacert/*cert-id*
iot:DescribeCertificate	DescribeCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:DescribeDefaultAuthorizer	DescribeDefaultAuthorizer	None
iot:DescribeEndpoint	DescribeEndpoint	*
iot:DescribeEventConfigurations	DescribeEventConfigurations	none
iot:DescribeIndex	DescribeIndex	arn:aws:iot: *region*:*account-id* :index/*index-name*
iot:DescribeJob	DescribeJob	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:DescribeJobExecution	DescribeJobExecution	None

Policy actions	AWS IoT API	Resources
iot:DescribeJobTemplate	DescribeJobTemplate	`arn:aws:iot: region:account-id :job/job-template-id`
iot:DescribeRoleAlias	DescribeRoleAlias	`arn:aws:iot: region:account-id :rolealias/ role-alias-name`
iot:DescribeThing	DescribeThing	`arn:aws:iot: region:account-id :thing/thing-name`
iot:DescribeThingGroup	DescribeThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
iot:DescribeThingRegistrationTask	DescribeThingRegistrationTask	None
iot:DescribeThingType	DescribeThingType	`arn:aws:iot: region:account-id :thingtype/ thing-type-name`

iot:DetachPolicy	DetachPolicy	`arn:aws:iot: region:account-id :cert/cert-id`
		or
		`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`

iot:DetachPrincipalPolicy	DetachPrincipalPolicy	`arn:aws:iot: region:account-id :cert/cert-id`
iot:DetachSecurityProfile	DetachSecurityProfile	`arn:aws:iot: region:account-id :securityprofile/ security-profile-name` `arn:aws:iot: region:account-id :dimension/ dimension-name`
iot:DetachThingPrincipal	DetachThingPrincipal	`arn:aws:iot: region:account-id :cert/cert-id`

Policy actions	AWS IoT API	Resources
iot:DisableTopicRule	DisableTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:EnableTopicRule	EnableTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:GetEffectivePolicies	GetEffectivePolicies	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:GetIndexingConfiguration	GetIndexingConfiguration	None
iot:GetJobDocument	GetJobDocument	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:GetLoggingOptions	GetLoggingOptions	*
iot:GetPolicy	GetPolicy	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:GetPolicyVersion	GetPolicyVersion	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:GetRegistrationCode	GetRegistrationCode	*
iot:GetTopicRule	GetTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:ListAttachedPolicies	ListAttachedPolicies	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name* or arn:aws:iot: *region*:*account-id* :cert/*cert-id*

Policy actions	AWS IoT API	Resources
iot:ListAuthorizers	ListAuthorizers	None
iot:ListCACertificates	ListCACertificates	*
iot:ListCertificates	ListCertificates	*
iot:ListCertificatesByCA	ListCertificatesByCA	*
iot:ListIndices	ListIndices	None
iot:ListJobExecutionsForJob	ListJobExecutionsForJob	None
iot:ListJobExecutionsForThing	ListJobExecutionsForThing	None
iot:ListJobs	ListJobs	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` if thingGroupName parameter used
iot:ListJobTemplates	ListJobs	None
iot:ListOutgoingCertificates	ListOutgoingCertificates	*
iot:ListPolicies	ListPolicies	*
iot:ListPolicyPrincipals	ListPolicyPrincipals	*

Policy actions	AWS IoT API	Resources
iot:ListPolicyVersions	ListPolicyVersions	`arn:aws:iot: region:account-id :policy/policy-name`
iot:ListPrincipalPolicies	ListPrincipalPolicies	`arn:aws:iot: region:account-id :cert/cert-id`
iot:ListPrincipalThings	ListPrincipalThings	`arn:aws:iot: region:account-id :cert/cert-id`
iot:ListRoleAliases	ListRoleAliases	None
iot:ListTargetsForPolicy	ListTargetsForPolicy	`arn:aws:iot: region:account-id :policy/policy-name`
iot:ListThingGroups	ListThingGroups	None
iot:ListThingGroupsForThing	ListThingGroupsForThing	`arn:aws:iot: region:account-id :thing/thing-name`
iot:ListThingPrincipals	ListThingPrincipals	`arn:aws:iot: region:account-id :thing/thing-name`
iot:ListThingRegistrationTaskReports	ListThingRegistrationTaskReports	None
iot:ListThingRegistrationTasks	ListThingRegistrationTasks	None
iot:ListThingTypes	ListThingTypes	*

Policy actions	AWS IoT API	Resources
iot:ListThings	ListThings	*
iot:ListThingsInThingGroup	ListThingsInThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
iot:ListTopicRules	ListTopicRules	*
iot:ListV2LoggingLevels	ListV2LoggingLevels	None
iot:RegisterCACertificate	RegisterCACertificate	*
iot:RegisterCertificate	RegisterCertificate	*
iot:RegisterThing	RegisterThing	None
iot:RejectCertificateTransfer	RejectCertificateTransfer	`arn:aws:iot: region:account-id :cert/cert-id`
iot:RemoveThingFromThingGroup	RemoveThingFromThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` `arn:aws:iot: region:account-id :thing/thing-name`
iot:ReplaceTopicRule	ReplaceTopicRule	`arn:aws:iot: region:account-id :rule/rule-name`
iot:SearchIndex	SearchIndex	`arn:aws:iot: region:account-id :index/index-id`
iot:SetDefaultAuthorizer	SetDefaultAuthorizer	`arn:aws:iot: region:account-id :authorizer/ authorizer-function-name`

Policy actions	AWS IoT API	Resources
iot:SetDefaultPolicyVersion	SetDefaultPolicyVersion	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:SetLoggingOptions	SetLoggingOptions	arn:aws:iot: *region*:*account-id* :role/*role-name*
iot:SetV2LoggingLevel	SetV2LoggingLevel	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*
iot:SetV2LoggingOptions	SetV2LoggingOptions	arn:aws:iot: *region*:*account-id* :role/*role-name*
iot:StartThingRegistrationTask	StartThingRegistrationTask	None
iot:StopThingRegistrationTask	StopThingRegistrationTask	None
iot:TestAuthorization	TestAuthorization	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:TestInvokeAuthorizer	TestInvokeAuthorizer	None
iot:TransferCertificate	TransferCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:UpdateAuthorizer	UpdateAuthorizer	arn:aws:iot: *region*:*account-id* :authorizerfunction/ *authorizer-function-name*
iot:UpdateCACertificate	UpdateCACertificate	arn:aws:iot: *region*:*account-id* :cacert/*cert-id*

Policy actions	AWS IoT API	Resources
iot:UpdateCertificate	UpdateCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:UpdateDimension	UpdateDimension	arn:aws:iot: *region*:*account-id* :dimension/ *dimension-name*
iot:UpdateEventConfigurations	UpdateEventConfigurations	None
iot:UpdateIndexingConfiguration	UpdateIndexingConfiguration	None
iot:UpdateRoleAlias	UpdateRoleAlias	arn:aws:iot: *region*:*account-id* :rolealias/ *role-alias-name*
iot:UpdateSecurityProfile	UpdateSecurityProfile	arn:aws:iot: *region*:*account-id* :securityprofile/ *security-profile-name* arn:aws:iot: *region*:*account-id* :dimension/ *dimension-name*
iot:UpdateThing	UpdateThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:UpdateThingGroup	UpdateThingGroup	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*
iot:UpdateThingGroupsForThing	UpdateThingGroupsForThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name* arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*

Policy actions in AWS IoT use the following prefix before the action: `iot:`. For example, to grant someone permission to list all IoT things registered in their AWS account with the `ListThings`

API, you include the `iot:ListThings` action in their policy. Policy statements must include either an `Action` or `NotAction` element. AWS IoT defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "ec2:action1",
      "ec2:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "iot:Describe*"
```

To see a list of AWS IoT actions, see Actions Defined by AWS IoT in the *IAM User Guide*.

Device Advisor actions

The following table lists the IAM IoT Device Advisor actions, the associated AWS IoT Device Advisor API, and the resource the action manipulates.

Policy actions	AWS IoT API	Resources
iotdeviceadvisor:CreateSuiteDefinition	CreateSuiteDefinition	None
iotdeviceadvisor:DeleteSuiteDefinition	DeleteSuiteDefinition	`arn:aws:iotdeviceadvisor: region:account-id :suitedefinition/ suite-definition-id`
iotdeviceadvisor:GetSuiteDefinition	GetSuiteDefinition	`arn:aws:iotdeviceadvisor: region:account-id :suitedefinition/ suite-definition-id`

Policy actions	AWS IoT API	Resources
iotdevice advisor:G etSuiteRun	GetSuiteRun	arn:aws:iotdeviceadvisor: *region*:*account-id* :suitedefinition/ *suite-run-id*
iotdevice advisor:G etSuiteRu nReport	GetSuiteR unReport	arn:aws:iotdeviceadvisor: *region*:*account-id* :suiterun/ *suite-definition-id* /*suite-run-id*
iotdevice advisor:L istSuiteD efinitions	ListSuite Definitions	None
iotdevice advisor:L istSuiteRuns	ListSuiteRuns	arn:aws:iotdeviceadvisor: *region*:*account-id* :suitedefinition/ *suite-definition-id*
iotdevice advisor:L istTagsFo rResource	ListTagsF orResource	arn:aws:iotdeviceadvisor: *region*:*account-id* :suitedefinition/ *suite-definition-id* arn:aws:iotdeviceadvisor: *region*:*account-id* :suiterun/suite-definition-id/ *suite-run-id*
iotdevice advisor:S tartSuiteRun	StartSuiteRun	arn:aws:iotdeviceadvisor: *region*:*account-id* :suitedefinition/ *suite-definition-id*
iotdevice advisor:T agResource	TagResource	arn:aws:iotdeviceadvisor: *region*:*account-id* :suitedefinition/ *suite-definition-id* arn:aws:iotdeviceadvisor: *region*:*account-id* :suiterun/suite-definition-id/ *suite-run-id*

Policy actions	AWS IoT API	Resources

iotdevice advisor:U ntagResource	UntagResource	`arn:aws:iotdeviceadvisor: region:account-id :suitedefinition/ suite-definition-id`
		
		`arn:aws:iotdeviceadvisor: region:account-id :suiterun/suite-definition-id/ suite-run-id`

iotdevice advisor:U pdateSuit eDefinition	UpdateSui teDefinition	`arn:aws:iotdeviceadvisor: region:account-id :suitedefinition/ suite-definition-id`
iotdevice advisor:S topSuiteRun	StopSuiteRun	`arn:aws:iotdeviceadvisor: region:account-id :suiterun/suite-definition-id/ suite-run-id`

Policy actions in AWS IoT Device Advisor use the following prefix before the action: `iotdeviceadvisor:`. For example, to grant someone permission to list all suite definitions registered in their AWS account with the ListSuiteDefinitions API, you include the `iotdeviceadvisor:ListSuiteDefinitions` action in their policy.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

AWS IoT resources

Policy actions	AWS IoT API	Resources
iot:AcceptCertificateTransfer	AcceptCertificateTransfer	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id* **Note** The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:AddThingToThingGroup	AddThingToThingGroup	`arn:aws:iot:` *region*:*account-id* `:thinggroup/` *thing-group-name* `arn:aws:iot:` *region*:*account-id* `:thing/`*thing-name*
iot:AssociateTargetsWithJob	AssociateTargetsWithJob	None
iot:AttachPolicy	AttachPolicy	`arn:aws:iot:` *region*:*account-id* `:thinggroup/` *thing-group-name* or `arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*
iot:AttachPrincipalPolicy	AttachPrincipalPolicy	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*
iot:AttachThingPrincipal	AttachThingPrincipal	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*
iot:CancelCertificateTransfer	CancelCertificateTransfer	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*

Policy actions	AWS IoT API	Resources
		Note The AWS account specified in the ARN must be the account to which the certificate is being transferred.
iot:CancelJob	CancelJob	`arn:aws:iot: region:account-id :job/job-id`

iot:CancelJobExecution	CancelJobExecution	`arn:aws:iot: region:account-id :job/job-id`
		
		`arn:aws:iot: region:account-id :thing/thing-name`

iot:ClearDefaultAuthorizer	ClearDefaultAuthorizer	None
iot:CreateAuthorizer	CreateAuthorizer	`arn:aws:iot: region:account-id :authorizer/ authorizer-function-name`
iot:CreateCertificateFromCsr	CreateCertificateFromCsr	*

iot:CreateJob	CreateJob	`arn:aws:iot: region:account-id :job/job-id`
		
		`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
		
		`arn:aws:iot: region:account-id :thing/thing-name`
		
		`arn:aws:iot: region:account-id :jobtemplate/ job-template-id`

Policy actions	AWS IoT API	Resources

iot:CreateJobTemplate	CreateJobTemplate	`arn:aws:iot: region:account-id :job/job-id`
		
		`arn:aws:iot: region:account-id :jobtemplate/ job-template-id`

iot:CreateKeysAndCertificate	CreateKeysAndCertificate	*
iot:CreatePolicy	CreatePolicy	`arn:aws:iot: region:account-id :policy/policy-name`
CreatePolicyVersion	iot:CreatePolicyVersion	`arn:aws:iot: region:account-id :policy/policy-name` **Note** This must be an AWS IoT policy, not an IAM policy.
iot:CreateRoleAlias	CreateRoleAlias	(parameter: roleAlias) `arn:aws:iot: region:account-id :rolealias/ role-alias-name`
iot:CreateThing	CreateThing	`arn:aws:iot: region:account-id :thing/thing-name`
iot:CreateThingGroup	CreateThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` for group being created and for parent group, if used
iot:CreateThingType	CreateThingType	`arn:aws:iot: region:account-id :thingtype/ thing-type-name`

Policy actions	AWS IoT API	Resources
iot:CreateTopicRule	CreateTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:DeleteAuthorizer	DeleteAuthorizer	arn:aws:iot: *region*:*account-id* :authorizer/ *authorizer-name*
iot:DeleteCACertificate	DeleteCACertificate	arn:aws:iot: *region*:*account-id* :cacert/*cert-id*
iot:DeleteCertificate	DeleteCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:DeleteJob	DeleteJob	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:DeleteJobExecution	DeleteJobExecution	arn:aws:iot: *region*:*account-id* :job/*job-id* arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:DeleteJobTemplate	DeleteJobTemplate	arn:aws:iot: *region*:*account-id* :jobtemplate/ *job-template-id*
iot:DeletePolicy	DeletePolicy	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:DeletePolicyVersion	DeletePolicyVersion	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:DeleteRegistrationCode	DeleteRegistrationCode	*
iot:DeleteRoleAlias	DeleteRoleAlias	arn:aws:iot: *region*:*account-id* :rolealias/ *role-alias-name*
iot:DeleteThing	DeleteThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name*

Policy actions	AWS IoT API	Resources
iot:DeleteThingGroup	DeleteThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
iot:DeleteThingType	DeleteThingType	`arn:aws:iot: region:account-id :thingtype/ thing-type-name`
iot:DeleteTopicRule	DeleteTopicRule	`arn:aws:iot: region:account-id :rule/rule-name`
iot:DeleteV2LoggingLevel	DeleteV2LoggingLevel	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
iot:DeprecateThingType	DeprecateThingType	`arn:aws:iot: region:account-id :thingtype/ thing-type-name`
iot:DescribeAuthorizer	DescribeAuthorizer	`arn:aws:iot: region:account-id :authorizer/ authorizer-function-name` (parameter: authorizerName) none
iot:DescribeCACertificate	DescribeCACertificate	`arn:aws:iot: region:account-id :cacert/cert-id`
iot:DescribeCertificate	DescribeCertificate	`arn:aws:iot: region:account-id :cert/cert-id`
iot:DescribeDefaultAuthorizer	DescribeDefaultAuthorizer	None
iot:DescribeEndpoint	DescribeEndpoint	*

Policy actions	AWS IoT API	Resources
iot:DescribeEventConfigurations	DescribeEventConfigurations	none
iot:DescribeIndex	DescribeIndex	arn:aws:iot: *region*:*account-id* :index/*index-name*
iot:DescribeJob	DescribeJob	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:DescribeJobExecution	DescribeJobExecution	None
iot:DescribeJobTemplate	DescribeJobTemplate	arn:aws:iot: *region*:*account-id* :jobtemplate/ *job-template-id*
iot:DescribeRoleAlias	DescribeRoleAlias	arn:aws:iot: *region*:*account-id* :rolealias/ *role-alias-name*
iot:DescribeThing	DescribeThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:DescribeThingGroup	DescribeThingGroup	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*
iot:DescribeThingRegistrationTask	DescribeThingRegistrationTask	None
iot:DescribeThingType	DescribeThingType	arn:aws:iot: *region*:*account-id* :thingtype/ *thing-type-name*
iot:DetachPolicy	DetachPolicy	arn:aws:iot: *region*:*account-id* :cert/*cert-id* or arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*

Policy actions	AWS IoT API	Resources
iot:DetachPrincipalPolicy	DetachPrincipalPolicy	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:DetachThingPrincipal	DetachThingPrincipal	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:DisableTopicRule	DisableTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:EnableTopicRule	EnableTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*
iot:GetEffectivePolicies	GetEffectivePolicies	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:GetIndexingConfiguration	GetIndexingConfiguration	None
iot:GetJobDocument	GetJobDocument	arn:aws:iot: *region*:*account-id* :job/*job-id*
iot:GetLoggingOptions	GetLoggingOptions	*
iot:GetPolicy	GetPolicy	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:GetPolicyVersion	GetPolicyVersion	arn:aws:iot: *region*:*account-id* :policy/*policy-name*
iot:GetRegistrationCode	GetRegistrationCode	*
iot:GetTopicRule	GetTopicRule	arn:aws:iot: *region*:*account-id* :rule/*rule-name*

Policy actions	AWS IoT API	Resources

iot:ListAttachedPolicies	ListAttachedPolicies	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
		
		or
		
		`arn:aws:iot: region:account-id :cert/cert-id`

iot:ListAuthorizers	ListAuthorizers	None
iot:ListCACertificates	ListCACertificates	*
iot:ListCertificates	ListCertificates	*
iot:ListCertificatesByCA	ListCertificatesByCA	*
iot:ListIndices	ListIndices	None
iot:ListJobExecutionsForJob	ListJobExecutionsForJob	None
iot:ListJobExecutionsForThing	ListJobExecutionsForThing	None
iot:ListJobs	ListJobs	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name` if thingGroupName parameter used
iot:ListJobTemplates	ListJobTemplates	None

Policy actions	AWS IoT API	Resources
iot:ListOutgoingCertificates	ListOutgoingCertificates	*
iot:ListPolicies	ListPolicies	*
iot:ListPolicyPrincipals	ListPolicyPrincipals	`arn:aws:iot:` *region*:*account-id* `:policy/`*policy-name*
iot:ListPolicyVersions	ListPolicyVersions	`arn:aws:iot:` *region*:*account-id* `:policy/`*policy-name*
iot:ListPrincipalPolicies	ListPrincipalPolicies	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*
iot:ListPrincipalThings	ListPrincipalThings	`arn:aws:iot:` *region*:*account-id* `:cert/`*cert-id*
iot:ListRoleAliases	ListRoleAliases	None
iot:ListTargetsForPolicy	ListTargetsForPolicy	`arn:aws:iot:` *region*:*account-id* `:policy/`*policy-name*
iot:ListThingGroups	ListThingGroups	None
iot:ListThingGroupsForThing	ListThingGroupsForThing	`arn:aws:iot:` *region*:*account-id* `:thing/`*thing-name*
iot:ListThingPrincipals	ListThingPrincipals	`arn:aws:iot:` *region*:*account-id* `:thing/`*thing-name*

Policy actions	AWS IoT API	Resources
iot:ListThingRegistrationTaskReports	ListThingRegistrationTaskReports	None
iot:ListThingRegistrationTasks	ListThingRegistrationTasks	None
iot:ListThingTypes	ListThingTypes	*
iot:ListThings	ListThings	*
iot:ListThingsInThingGroup	ListThingsInThingGroup	`arn:aws:iot:` *`region`*`:`*`account-id`*` :thinggroup/ `*`thing-group-name`*
iot:ListTopicRules	ListTopicRules	*
iot:ListV2LoggingLevels	ListV2LoggingLevels	None
iot:RegisterCACertificate	RegisterCACertificate	*
iot:RegisterCertificate	RegisterCertificate	*
iot:RegisterThing	RegisterThing	None
iot:RejectCertificateTransfer	RejectCertificateTransfer	`arn:aws:iot:` *`region`*`:`*`account-id`*` :cert/`*`cert-id`*

Policy actions	AWS IoT API	Resources

iot:RemoveThingFromThingGroup	RemoveThingFromThingGroup	`arn:aws:iot: region:account-id :thinggroup/ thing-group-name`
		
		`arn:aws:iot: region:account-id :thing/thing-name`

iot:ReplaceTopicRule	ReplaceTopicRule	`arn:aws:iot: region:account-id :rule/rule-name`
iot:SearchIndex	SearchIndex	`arn:aws:iot: region:account-id :index/index-id`
iot:SetDefaultAuthorizer	SetDefaultAuthorizer	`arn:aws:iot: region:account-id :authorizer/ authorizer-function-name`
iot:SetDefaultPolicyVersion	SetDefaultPolicyVersion	`arn:aws:iot: region:account-id :policy/policy-name`
iot:SetLoggingOptions	SetLoggingOptions	*
iot:SetV2LoggingLevel	SetV2LoggingLevel	*
iot:SetV2LoggingOptions	SetV2LoggingOptions	*
iot:StartThingRegistrationTask	StartThingRegistrationTask	None
iot:StopThingRegistrationTask	StopThingRegistrationTask	None

Policy actions	AWS IoT API	Resources
iot:TestAuthorization	TestAuthorization	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:TestInvokeAuthorizer	TestInvokeAuthorizer	None
iot:TransferCertificate	TransferCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:UpdateAuthorizer	UpdateAuthorizer	arn:aws:iot: *region*:*account-id* :authorizerfunction/ *authorizer-function-name*
iot:UpdateCACertificate	UpdateCACertificate	arn:aws:iot: *region*:*account-id* :cacert/*cert-id*
iot:UpdateCertificate	UpdateCertificate	arn:aws:iot: *region*:*account-id* :cert/*cert-id*
iot:UpdateEventConfigurations	UpdateEventConfigurations	None
iot:UpdateIndexingConfiguration	UpdateIndexingConfiguration	None
iot:UpdateRoleAlias	UpdateRoleAlias	arn:aws:iot: *region*:*account-id* :rolealias/ *role-alias-name*
iot:UpdateThing	UpdateThing	arn:aws:iot: *region*:*account-id* :thing/*thing-name*
iot:UpdateThingGroup	UpdateThingGroup	arn:aws:iot: *region*:*account-id* :thinggroup/ *thing-group-name*

Policy actions	AWS IoT API	Resources
iot:UpdateThingGroupsForThing	UpdateThingGroupsForThing	`arn:aws:iot: region:account-id :thing/thing-name`

For more information about the format of ARNs, see Amazon Resource Names (ARNs) and AWS Service Namespaces.

Some AWS IoT actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

To see a list of AWS IoT resource types and their ARNs, see Resources Defined by AWS IoT in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS IoT.

Device Advisor resources

To define resource-level restrictions for AWS IoT Device Advisor IAM policies, use the following resource ARN formats for suite definitions and suite runs.

Suite definition resource ARN format

`arn:aws:iotdeviceadvisor:region:account-id:suitedefinition/suite-definition-id`

Suite run resource ARN format

`arn:aws:iotdeviceadvisor:region:account-id:suiterun/suite-definition-id/suite-run-id`

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

AWS IoT defines its own set of condition keys and also supports using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*.

AWS IoT condition keys

AWS IoT condition keys	Description	Type
`aws:RequestTag/${tag-key}`	A tag key that is present in the request that the user makes to AWS IoT.	String
`aws:ResourceTag/${tag-key}`	The tag key component of a tag attached to an AWS IoT resource.	String
`aws:TagKeys`	The list of all the tag key names associated with the resource in the request.	String

To see a list of AWS IoT condition keys, see Condition Keys for AWS IoT in the *IAM User Guide*. To learn with which actions and resources you can use a condition key, see Actions Defined by AWS IoT.

Examples

To view examples of AWS IoT identity-based policies, see AWS IoT identity-based policy examples.

AWS IoT resource-based policies

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on the AWS IoT resource and under what conditions.

AWS IoT does not support IAM resource-based policies. It does, however, support AWS IoT resource-based policies. For more information, see AWS IoT Core policies.

Authorization based on AWS IoT tags

You can attach tags to AWS IoT resources or pass tags in a request to AWS IoT. To control access based on tags, you provide tag information in the condition element of a policy using the `iot:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys. For more information, see Using tags with IAM policies. For more information about tagging AWS IoT resources, see Tagging your AWS IoT resources.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Viewing AWS IoT resources based on tags.

AWS IoT IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

AWS IoT supports using temporary credentials.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

AWS IoT does not supports service-linked roles.

Service roles

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your

IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

AWS IoT identity-based policy examples

By default, IAM users and roles don't have permission to create or modify AWS IoT resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete AWS IoT resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions –** To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions –** When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more

information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the AWS IoT console

To access the AWS IoT console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS IoT resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

To ensure that those entities can still use the AWS IoT console, also attach the following AWS managed policy to the entities: `AWSIoTFullAccess`. For more information, see Adding Permissions to a User in the *IAM User Guide*.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Viewing AWS IoT resources based on tags

You can use conditions in your identity-based policy to control access to AWS IoT resources based on tags. This example shows how you might create a policy that allows viewing a thing. However, permission is granted only if the thing tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListBillingGroupsInConsole",
            "Effect": "Allow",
            "Action": "iot:ListBillingGroups",
            "Resource": "*"
        },
        {
            "Sid": "ViewBillingGroupsIfOwner",
            "Effect": "Allow",
           "Action": "iot:DescribeBillingGroup",
            "Resource": "arn:aws:iot:*:*:billinggroup/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
        }
        }
    ]
}
```

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view an AWS IoT billing group, the billing group must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise, he is denied access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see [IAM JSON Policy Elements: Condition](#) in the *IAM User Guide*.

Viewing AWS IoT Device Advisor resources based on tags

You can use conditions in your identity-based policy to control access to AWS IoT Device Advisor resources based on tags. The following example shows how you can create a policy that allows viewing a particular suite definition. However, permission is granted only if the suite definition tag has `SuiteType` set to the value of `MQTT`. This policy also grants the permissions necessary to complete this action on the console.

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ViewSuiteDefinition",
            "Effect": "Allow",
            "Action": "iotdeviceadvisor:GetSuiteDefinition",
            "Resource": "arn:aws:iotdeviceadvisor:*:*:suitedefinition/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/SuiteType": "MQTT"}
        }
        }
    ]
}
```

AWS managed policies for AWS IoT

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the **ReadOnlyAccess** AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

> **Note**
>
> AWS IoT works with both AWS IoT and IAM policies. This topic discusses only IAM policies, which defines a policy action for control plane and data plane API operations. See also AWS IoT Core policies.

AWS managed policy: AWSIoTConfigAccess

You can attach the `AWSIoTConfigAccess` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to all AWS IoT configuration operations. This policy can affect data processing and storage. To view this policy in the AWS Management Console, see AWSIoTConfigAccess.

Permissions details

This policy includes the following permissions.

- `iot` – Retrieve AWS IoT data and perform IoT configuration actions.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:AcceptCertificateTransfer",
                "iot:AddThingToThingGroup",
                "iot:AssociateTargetsWithJob",
                "iot:AttachPolicy",
                "iot:AttachPrincipalPolicy",
                "iot:AttachThingPrincipal",
                "iot:CancelCertificateTransfer",
```

```
                "iot:CancelJob",
                "iot:CancelJobExecution",
                "iot:ClearDefaultAuthorizer",
                "iot:CreateAuthorizer",
                "iot:CreateCertificateFromCsr",
                "iot:CreateJob",
                "iot:CreateKeysAndCertificate",
                "iot:CreateOTAUpdate",
                "iot:CreatePolicy",
                "iot:CreatePolicyVersion",
                "iot:CreateRoleAlias",
                "iot:CreateStream",
                "iot:CreateThing",
                "iot:CreateThingGroup",
                "iot:CreateThingType",
                "iot:CreateTopicRule",
                "iot:DeleteAuthorizer",
                "iot:DeleteCACertificate",
                "iot:DeleteCertificate",
                "iot:DeleteJob",
                "iot:DeleteJobExecution",
                "iot:DeleteOTAUpdate",
                "iot:DeletePolicy",
                "iot:DeletePolicyVersion",
                "iot:DeleteRegistrationCode",
                "iot:DeleteRoleAlias",
                "iot:DeleteStream",
                "iot:DeleteThing",
                "iot:DeleteThingGroup",
                "iot:DeleteThingType",
                "iot:DeleteTopicRule",
                "iot:DeleteV2LoggingLevel",
                "iot:DeprecateThingType",
                "iot:DescribeAuthorizer",
                "iot:DescribeCACertificate",
                "iot:DescribeCertificate",
                "iot:DescribeDefaultAuthorizer",
                "iot:DescribeEndpoint",
                "iot:DescribeEventConfigurations",
                "iot:DescribeIndex",
                "iot:DescribeJob",
                "iot:DescribeJobExecution",
                "iot:DescribeRoleAlias",
                "iot:DescribeStream",
```

```
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:DescribeThingRegistrationTask",
                "iot:DescribeThingType",
                "iot:DetachPolicy",
                "iot:DetachPrincipalPolicy",
                "iot:DetachThingPrincipal",
                "iot:DisableTopicRule",
                "iot:EnableTopicRule",
                "iot:GetEffectivePolicies",
                "iot:GetIndexingConfiguration",
                "iot:GetJobDocument",
                "iot:GetLoggingOptions",
                "iot:GetOTAUpdate",
                "iot:GetPolicy",
                "iot:GetPolicyVersion",
                "iot:GetRegistrationCode",
                "iot:GetTopicRule",
                "iot:GetV2LoggingOptions",
                "iot:ListAttachedPolicies",
                "iot:ListAuthorizers",
                "iot:ListCACertificates",
                "iot:ListCertificates",
                "iot:ListCertificatesByCA",
                "iot:ListIndices",
                "iot:ListJobExecutionsForJob",
                "iot:ListJobExecutionsForThing",
                "iot:ListJobs",
                "iot:ListOTAUpdates",
                "iot:ListOutgoingCertificates",
                "iot:ListPolicies",
                "iot:ListPolicyPrincipals",
                "iot:ListPolicyVersions",
                "iot:ListPrincipalPolicies",
                "iot:ListPrincipalThings",
                "iot:ListRoleAliases",
                "iot:ListStreams",
                "iot:ListTargetsForPolicy",
                "iot:ListThingGroups",
                "iot:ListThingGroupsForThing",
                "iot:ListThingPrincipals",
                "iot:ListThingRegistrationTaskReports",
                "iot:ListThingRegistrationTasks",
                "iot:ListThings",
```

```
                "iot:ListThingsInThingGroup",
                "iot:ListThingTypes",
                "iot:ListTopicRules",
                "iot:ListV2LoggingLevels",
                "iot:RegisterCACertificate",
                "iot:RegisterCertificate",
                "iot:RegisterThing",
                "iot:RejectCertificateTransfer",
                "iot:RemoveThingFromThingGroup",
                "iot:ReplaceTopicRule",
                "iot:SearchIndex",
                "iot:SetDefaultAuthorizer",
                "iot:SetDefaultPolicyVersion",
                "iot:SetLoggingOptions",
                "iot:SetV2LoggingLevel",
                "iot:SetV2LoggingOptions",
                "iot:StartThingRegistrationTask",
                "iot:StopThingRegistrationTask",
                "iot:TestAuthorization",
                "iot:TestInvokeAuthorizer",
                "iot:TransferCertificate",
                "iot:UpdateAuthorizer",
                "iot:UpdateCACertificate",
                "iot:UpdateCertificate",
                "iot:UpdateEventConfigurations",
                "iot:UpdateIndexingConfiguration",
                "iot:UpdateRoleAlias",
                "iot:UpdateStream",
                "iot:UpdateThing",
                "iot:UpdateThingGroup",
                "iot:UpdateThingGroupsForThing",
                "iot:UpdateAccountAuditConfiguration",
                "iot:DescribeAccountAuditConfiguration",
                "iot:DeleteAccountAuditConfiguration",
                "iot:StartOnDemandAuditTask",
                "iot:CancelAuditTask",
                "iot:DescribeAuditTask",
                "iot:ListAuditTasks",
                "iot:CreateScheduledAudit",
                "iot:UpdateScheduledAudit",
                "iot:DeleteScheduledAudit",
                "iot:DescribeScheduledAudit",
                "iot:ListScheduledAudits",
                "iot:ListAuditFindings",
```

```
                "iot:CreateSecurityProfile",
                "iot:DescribeSecurityProfile",
                "iot:UpdateSecurityProfile",
                "iot:DeleteSecurityProfile",
                "iot:AttachSecurityProfile",
                "iot:DetachSecurityProfile",
                "iot:ListSecurityProfiles",
                "iot:ListSecurityProfilesForTarget",
                "iot:ListTargetsForSecurityProfile",
                "iot:ListActiveViolations",
                "iot:ListViolationEvents",
                "iot:ValidateSecurityProfileBehaviors"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSIoTConfigReadOnlyAccess

You can attach the `AWSIoTConfigReadOnlyAccess` policy to your IAM identities.

This policy grants the associated identity permissions that allow read-only access to all AWS IoT configuration operations. To view this policy in the AWS Management Console, see AWSIoTConfigReadOnlyAccess.

Permissions details

This policy includes the following permissions.

- `iot` – Perform read-only operations of IoT configuration actions.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:DescribeAuthorizer",
                "iot:DescribeCACertificate",
                "iot:DescribeCertificate",
                "iot:DescribeDefaultAuthorizer",
                "iot:DescribeEndpoint",
                "iot:DescribeEventConfigurations",
                "iot:DescribeIndex",
                "iot:DescribeJob",
                "iot:DescribeJobExecution",
                "iot:DescribeRoleAlias",
                "iot:DescribeStream",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:DescribeThingRegistrationTask",
                "iot:DescribeThingType",
                "iot:GetEffectivePolicies",
                "iot:GetIndexingConfiguration",
                "iot:GetJobDocument",
                "iot:GetLoggingOptions",
                "iot:GetOTAUpdate",
                "iot:GetPolicy",
                "iot:GetPolicyVersion",
                "iot:GetRegistrationCode",
                "iot:GetTopicRule",
                "iot:GetV2LoggingOptions",
                "iot:ListAttachedPolicies",
                "iot:ListAuthorizers",
                "iot:ListCACertificates",
                "iot:ListCertificates",
                "iot:ListCertificatesByCA",
                "iot:ListIndices",
                "iot:ListJobExecutionsForJob",
                "iot:ListJobExecutionsForThing",
                "iot:ListJobs",
                "iot:ListOTAUpdates",
                "iot:ListOutgoingCertificates",
                "iot:ListPolicies",
                "iot:ListPolicyPrincipals",
                "iot:ListPolicyVersions",
                "iot:ListPrincipalPolicies",
                "iot:ListPrincipalThings",
                "iot:ListRoleAliases",
```

```
                "iot:ListStreams",
                "iot:ListTargetsForPolicy",
                "iot:ListThingGroups",
                "iot:ListThingGroupsForThing",
                "iot:ListThingPrincipals",
                "iot:ListThingRegistrationTaskReports",
                "iot:ListThingRegistrationTasks",
                "iot:ListThings",
                "iot:ListThingsInThingGroup",
                "iot:ListThingTypes",
                "iot:ListTopicRules",
                "iot:ListV2LoggingLevels",
                "iot:SearchIndex",
                "iot:TestAuthorization",
                "iot:TestInvokeAuthorizer",
                "iot:DescribeAccountAuditConfiguration",
                "iot:DescribeAuditTask",
                "iot:ListAuditTasks",
                "iot:DescribeScheduledAudit",
                "iot:ListScheduledAudits",
                "iot:ListAuditFindings",
                "iot:DescribeSecurityProfile",
                "iot:ListSecurityProfiles",
                "iot:ListSecurityProfilesForTarget",
                "iot:ListTargetsForSecurityProfile",
                "iot:ListActiveViolations",
                "iot:ListViolationEvents",
                "iot:ValidateSecurityProfileBehaviors"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSIoTDataAccess

You can attach the `AWSIoTDataAccess` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to all AWS IoT data operations. Data operations send data over MQTT or HTTP protocols. To view this policy in the AWS Management Console, see `AWSIoTDataAccess`.

Permissions details

This policy includes the following permissions.

- `iot` – Retrieve AWS IoT data and allow full access to AWS IoT messaging actions.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect",
                "iot:Publish",
                "iot:Subscribe",
                "iot:Receive",
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DeleteThingShadow",
                "iot:ListNamedShadowsForThing"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSIoTFullAccess

You can attach the `AWSIoTFullAccess` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to all AWS IoT configuration and messaging operations. To view this policy in the AWS Management Console, see AWSIoTFullAccess.

Permissions details

This policy includes the following permissions.

- `iot` – Retrieve AWS IoT data and allow full access to AWS IoT configuration and messaging actions.
- `iotjobsdata` – Retrieve AWS IoT Jobs data and allow full access to AWS IoT Jobs data plane API operations.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:*",
                "iotjobsdata:*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSIoTLogging

You can attach the `AWSIoTLogging` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to create Amazon CloudWatch Logs groups and stream logs to the groups. This policy is attached to your CloudWatch logging role. To view this policy in the AWS Management Console, see AWSIoTLogging.

Permissions details

This policy includes the following permissions.

- `logs` – Retrieve CloudWatch logs. Also allows creation of CloudWatch Logs groups and stream logs to the groups.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:PutMetricFilter",
                "logs:PutRetentionPolicy",
                "logs:GetLogEvents",
                "logs:DeleteLogStream"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

AWS managed policy: AWSIoTOTAUpdate

You can attach the `AWSIoTOTAUpdate` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to create AWS IoT jobs, AWS IoT code signing jobs, and to describe AWS code signer jobs. To view this policy in the AWS Management Console, see [AWSIoTOTAUpdate.](AWSIoTOTAUpdate)

Permissions details

This policy includes the following permissions.

- `iot` – Create AWS IoT jobs and code signing jobs.
- `signer` – Perform creation of AWS code signer jobs.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "iot:CreateJob",
            "signer:DescribeSigningJob"
        ],
        "Resource": "*"
    }
}
```

AWS managed policy: AWSIoTRuleActions

You can attach the `AWSIoTRuleActions` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to all AWS services supported in AWS IoT rule actions. To view this policy in the AWS Management Console, see AWSIoTRuleActions.

Permissions details

This policy includes the following permissions.

- `iot` - Perform actions for publishing rule action messages.

- `dynamodb` - Insert a message into a DynamoDB table or split a message into multiple columns of a DynamoDB table.
- `s3` - Store an object in an Amazon S3 bucket.
- `kinesis` - Send a message to an Amazon Kinesis stream object.
- `firehose` - Insert a record in a Firehose stream object.
- `cloudwatch` - Change CloudWatch alarm state or send message data to CloudWatch metric.
- `sns` - Perform operation to publish a notification using Amazon SNS. This operation is scoped to AWS IoT SNS topics.
- `sqs` - Insert a message to add to the SQS queue.
- `es` - Send a message to the OpenSearch Service service.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "dynamodb:PutItem",
            "kinesis:PutRecord",
            "iot:Publish",
            "s3:PutObject",
            "sns:Publish",
            "sqs:SendMessage*",
            "cloudwatch:SetAlarmState",
            "cloudwatch:PutMetricData",
            "es:ESHttpPut",
            "firehose:PutRecord"
        ],
        "Resource": "*"
    }
}
```

AWS managed policy: AWSIoTThingsRegistration

You can attach the `AWSIoTThingsRegistration` policy to your IAM identities.

This policy grants the associated identity permissions that allow access to register things in bulk using the `StartThingRegistrationTask` API. This policy can affect data processing and storage. To view this policy in the AWS Management Console, see AWSIoTThingsRegistration.

Permissions details

This policy includes the following permissions.

- `iot` - Perform actions for creating things and attaching policies and certificates when registering in bulk.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:AddThingToThingGroup",
                "iot:AttachPolicy",
                "iot:AttachPrincipalPolicy",
                "iot:AttachThingPrincipal",
                "iot:CreateCertificateFromCsr",
                "iot:CreatePolicy",
                "iot:CreateThing",
                "iot:DescribeCertificate",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:DescribeThingType",
                "iot:DetachPolicy",
                "iot:DetachThingPrincipal",
                "iot:GetPolicy",
                "iot:ListAttachedPolicies",
                "iot:ListPolicyPrincipals",
                "iot:ListPrincipalPolicies",
                "iot:ListPrincipalThings",
                "iot:ListTargetsForPolicy",
                "iot:ListThingGroupsForThing",
                "iot:ListThingPrincipals",
```

```
                "iot:RegisterCertificate",
                "iot:RegisterThing",
                "iot:RemoveThingFromThingGroup",
                "iot:UpdateCertificate",
                "iot:UpdateThing",
                "iot:UpdateThingGroupsForThing",
                "iot:AddThingToBillingGroup",
                "iot:DescribeBillingGroup",
                "iot:RemoveThingFromBillingGroup"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

AWS IoT updates to AWS managed policies

View details about updates to AWS managed policies for AWS IoT since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the AWS IoT Document history page.

Change	Description	Date
AWSIoTFullAccess – Update to an existing policy	AWS IoT added new permissions to allow users to access AWS IoT Jobs data plane API operations using the HTTP protocol. A new IAM policy prefix, `iotjobsdata:` , provides you finer grained access control to access AWS IoT Jobs data plane endpoints . For control plane API	May 11, 2022

Change	Description	Date
	operations, you still use the `iot:` prefix. For more information, see [AWS IoT Core policies for HTTPS protocol]().	
AWS IoT started tracking changes	AWS IoT started tracking changes for its AWS managed policies.	May 11, 2022

Troubleshooting AWS IoT identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT and IAM.

Topics

- [I am not authorized to perform an action in AWS IoT]()
- [I am not authorized to perform iam:PassRole]()
- [I want to allow people outside of my AWS account to access my AWS IoT resources]()

I am not authorized to perform an action in AWS IoT

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the IAM user, `mateojackson`, tries to use the console to view details about a thing resource but doesn't have the `iot:DescribeThing` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 iot:DescribeThing on resource: MyIoTThing
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the thing resource by using the `iot:DescribeThing` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

Using AWS IoT Device Advisor

If you're using AWS IoT Device Advisor, the following example error occurs when the user `mateojackson` tries to use the console to view details about a suite definition but doesn't have the `iotdeviceadvisor:GetSuiteDefinition` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 iotdeviceadvisor:GetSuiteDefinition on resource: MySuiteDefinition
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the `MySuiteDefinition` resource using the `iotdeviceadvisor:GetSuiteDefinition` action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS IoT.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS IoT. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS IoT resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS IoT supports these features, see How AWS IoT works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Logging and Monitoring

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. For information on logging and monitoring procedures, see Monitoring AWS IoT

Monitoring Tools

AWS provides tools that you can use to monitor AWS IoT. You can configure some of these tools to do the monitoring for you. Some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated Monitoring Tools

You can use the following automated monitoring tools to watch AWS IoT and report when something is wrong:

- **Amazon CloudWatch Alarms –** Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods. For more information, see Monitor AWS IoT alarms and metrics using Amazon CloudWatch.

- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. Amazon CloudWatch Logs also allows you to see critical steps AWS IoT Device Advisor test cases take, generated events and MQTT messages sent from your devices or AWS IoT Core during test execution. These logs make it possible to debug and take corrective actions on your devices. For more information, see Monitor AWS IoT using CloudWatch Logs For more information about using Amazon CloudWatch, see Monitoring Log Files in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see What Is Amazon CloudWatch Events in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Logging AWS IoT API calls using AWS CloudTrail and also Working with CloudTrail Log Files in the *AWS CloudTrail User Guide*.

Manual Monitoring Tools

Another important part of monitoring AWS IoT involves manually monitoring those items that the CloudWatch alarms don't cover. The AWS IoT, CloudWatch, and other AWS service console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on AWS IoT.

- AWS IoT dashboard shows:
  - CA certificates
  - Certificates
  - Polices
  - Rules
  - Things
- CloudWatch home page shows:
  - Current alarms and status.
  - Graphs of alarms and resources.
  - Service health status.

  You can use CloudWatch to do the following:

- Create customized dashboards to monitor the services you care about.
- Graph metric data to troubleshoot issues and discover trends.
- Search and browse all your AWS resource metrics.
- Create and edit alarms to be notified of problems.

Compliance validation for AWS IoT Core

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in AWS IoT Core

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

AWS IoT Core stores information about your devices in the device registry. It also stores CA certificates, device certificates, and device shadow data. In the event of hardware or network failures, this data is automatically replicated across Availability Zones but not across Regions.

AWS IoT Core publishes MQTT events when the device registry is updated. You can use these messages to back up your registry data and save it somewhere, like a DynamoDB table. You are responsible for saving certificates that AWS IoT Core creates for you or those you create yourself.

Device shadow stores state data about your devices and can be resent when a device comes back online. AWS IoT Device Advisor stores information about your test suite configuration. This data is automatically replicated in the event of hardware or network failures.

AWS IoT Core resources are Region-specific and aren't replicated across AWS Regions unless you specifically do so.

For information about Security best practices, see Security best practices in AWS IoT Core.

Using AWS IoT Core with interface VPC endpoints

With AWS IoT Core, you can create IoT control plane endpoints and IoT data endpoints within your virtual private cloud (VPC) by using interface VPC endpoints. Interface VPC endpoints are powered by AWS PrivateLink, an AWS technology that you can use to access services running on AWS by using private IP addresses. For more information, see Amazon Virtual Private Cloud.

To connect devices in the field on remote networks, such as a corporate network to your Amazon VPC, refer to the options listed in the Network-to-Amazon VPC connectivity matrix.

Contents

Creating VPC endpoints for AWS IoT Core control plane

You can create a VPC endpoint for AWS IoT Core control plane API to connect your devices to AWS IoT services and other AWS services. To get started with VPC endpoints, create an interface VPC endpoint and select AWS IoT Core as the AWS service. If you are using the CLI, first call describe-vpc-endpoint-services to ensure that you are choosing an Availability Zone where AWS IoT Core is present in your particular AWS Region. For example, in us-east-1, this command would look like:

```
aws ec2 describe-vpc-endpoint-services --service-name com.amazonaws.us-east-1.iot.api
```

See the detailed instructions below to Create an Amazon VPC interface endpoint for AWS IoT Core control plane.

Creating VPC endpoints for AWS IoT Core data plane

You can create a VPC endpoint for AWS IoT Core data plane API to connect your devices to AWS IoT services and other AWS services. To get started with VPC endpoints, create an interface VPC endpoint and select AWS IoT Core as the AWS service. If you are using the CLI, first call describe-vpc-endpoint-services to ensure that you are choosing an Availability Zone where AWS IoT Core is present in your particular AWS Region. For example, in us-east-1, this command would look like:

```
aws ec2 describe-vpc-endpoint-services --service-name com.amazonaws.us-east-1.iot.data
```

> **Note**
>
> The VPC feature for automatically creating a DNS record is disabled. To connect to these endpoints, you must manually create a Private DNS record. For more information about Private VPC DNS records, see Private DNS for interface endpoints. For more information about AWS IoT Core VPC limitations, see Limitations .

To connect MQTT clients to the VPC endpoint interfaces:

- You must manually create DNS records in a private hosted zone that is attached to your VPC. To get started, see Creating a private hosted zone.

- Within your private hosted zone, create an alias record for each elastic network interface IP for the VPC endpoint. If you have multiple network interface IPs for multiple VPC endpoints, create weighted DNS records with equal weights across all the weighted records. These IP addresses are available from the DescribeNetworkInterfaces API call when filtered by the VPC endpoint ID in the description field.

See the detailed instructions below to Create an Amazon VPC interface endpoint and Configure private hosted zone for AWS IoT Core data plane.

Creating VPC endpoints for AWS IoT Core credential provider

You can create a VPC endpoint for AWS IoT Core credential provider to connect devices using client certificate-based authentication and get temporary AWS credentials in AWS Signature Version 4 format. To get started with VPC endpoints for AWS IoT Core credential provider, run the create-vpc-endpoint CLI command to create an interface VPC endpoint and select AWS IoT Core credential provider as the AWS service. To ensure that you are choosing an Availability Zone where AWS IoT Core is present in your particular AWS Region, your first run the describe-vpc-endpoint-services command. For example, in us-east-1, this command would look like:

```
aws ec2 describe-vpc-endpoint-services --service-name com.amazonaws.us-
east-1.iot.credentials
```

Note

The VPC feature for automatically creating a DNS record is disabled. To connect to these endpoints, you must manually create a Private DNS record. For more information about Private VPC DNS records, see Private DNS for interface endpoints. For more information about AWS IoT Core VPC limitations, see Limitations .

To connect HTTP clients to the VPC endpoint interfaces:

- You must manually create DNS records in a private hosted zone that is attached to your VPC. To get started, see Creating A private hosted zone.

- Within your private hosted zone, create an alias record for each elastic network interface IP for the VPC endpoint. If you have multiple network interface IPs for multiple VPC endpoints, create weighted DNS records with equal weights across all the weighted records. These IP addresses are available from the DescribeNetworkInterfaces API call when filtered by the VPC endpoint ID in the description field.

See the detailed instructions below to Create an Amazon VPC interface endpoint and Configure private hosted zone for AWS IoT Core credential provider.

Creating an Amazon VPC interface endpoint

You can create an interface VPC endpoint to connect to AWS services powered by AWS PrivateLink. Use the following procedure to create an interface VPC endpoint that connects to AWS IoT Core data plane or AWS IoT Core credential provider. For more information, see Access an AWS service using an interface VPC endpoint.

> **Note**
>
> The processes to create an Amazon VPC interface endpoint for AWS IoT Core data plane and AWS IoT Core credential provider are similar, but you must make endpoint specific changes to make the connection work.

To create an interface VPC endpoint using VPC Endpoints console

1. Navigate to the VPC **Endpoints** console, under **Virtual private cloud** on the left menu, choose **Endpoints** then **Create Endpoint**.
2. In the **Create endpoint** page, specify the following information.
   - Choose **AWS services** for **Service category**.
   - For **Service Name**, search by entering the keyword `iot`. In the list of `iot` services displayed, choose the endpoint.

     If you create a VPC endpoint for AWS IoT Core control plane, choose the AWS IoT Core control plane API endpoint for your AWS Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot.api`.

     If you create a VPC endpoint for AWS IoT Core data plane, choose the AWS IoT Core data plane API endpoint for your Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot.data`.

     If you create a VPC endpoint for AWS IoT Core credential provider, choose the AWS IoT Core credential provider endpoint for your Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot.credentials`.

     If you create a VPC endpoint for Federal Information Processing Standard (FIPS) regions, choose the FIPS API endpoint for your AWS Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot-fips.api`. This is only for control plane.

> **Note**
>
> The service name for AWS IoT Core data plane in China Region will be of the format `cn.com.amazonaws.region.iot.data`. The service name for AWS IoT Core control plane in China Region will be of the format `com.amazonaws.region.iot.api`.

- For **VPC** and **Subnets**, choose the VPC where you want to create the endpoint, and the Availability Zones (AZs) in which you want to create the endpoint network.
- For **Enable DNS name**, make sure that **Enable for this endpoint** is not selected for AWS IoT Core data plane and AWS IoT Core credential provider. Neither AWS IoT Core data plane nor AWS IoT Core credential provider supports private DNS names yet.

  For AWS IoT Core control plane, **Enable DNS name** is selected by default. This ensures that any requests to the AWS IoT Core control plane public endpoints will route through the VPC endpoints instead. When this is enabled, you do not need to configure a privated hosted zone.
- For **Security group**, choose the security groups you want to associate with the endpoint network interfaces.
- Optionally, you can add or remove tags. Tags are name-value pairs that you use to associate with your endpoint.

3. To create your VPC endpoint, choose **Create endpoint**.

After you create the AWS PrivateLink endpoint, in the **Details** tab of your endpoint, you'll see a list of DNS names. You can use one of these DNS names you created in this section to configure your private hosted zone. If you are using AWS IoT Core control plane, you do not need to configure a private hosted zone.

Configure a private hosted zone

> **Note**
>
> If you are using AWS IoT Core control plane and have **Enable DNS name** selected, you do not need to configure a private hosted zone. If you disable it, you must follow this procedure to configure a private hosted zone.

You can use one of these DNS names you created in the previous section to configure your private hosted zone.

For AWS IoT Core data plane

The DNS name must be your domain configuration name or your `IoT:Data-ATS` endpoint. An example DNS name can be: `*xxx*-ats.data.iot.*region*.amazonaws.com`.

For AWS IoT Core credential provider

The DNS name must be your `iot:CredentialProvider` endpoint. An example DNS name can be: `*xxxx*.credentials.iot.*region*.amazonaws.com`.

For AWS IoT Core control plane

The DNS name must be your AWS IoT Core control plane endpoint. An example DNS name for AWS IoT Core control plane is `*xxxx*.api.iot.*region*.amazonaws.com`.

> **Note**
>
> The processes to configure private hosted zone for AWS IoT Core data plane and AWS IoT Core credential provider are similar, but you must make endpoint specific changes to make the connection work.

Create a private hosted zone

To create a private hosted zone using Route 53 console

1. Navigate to the Route 53 **Hosted zones** console and choose **Create hosted zone**.
2. In the **Create hosted zone** page, specify the following information.
   - For **Domain name**, enter the endpoint address for your `iot:Data-ATS` or `iot:CredentialProvider` endpoint. The following AWS CLI command shows how to get the endpoint through a public network: `aws iot describe-endpoint --endpoint-type iot:Data-ATS`, or `aws iot describe-endpoint --endpoint-type iot:CredentialProvider`.

> **Note**
>
> If you're using custom domains, see [Using custom domains with VPC endpoints](). Custom domains are not supported for AWS IoT Core credential provider.

- For **Type**, choose **Private hosted zone**.
- Optionally, you can add or remove tags to associate with your hosted zone.

3. To create your private hosted zone, choose **Create hosted zone**.

For more information, see [Creating a private hosted zone]().

Create a record

After you have created a private hosted zone, you can create a record that tells the DNS how you want traffic to be routed to that domain.

To create a record

1. In the list of hosted zones displayed, choose the private hosted zone that you created earlier and choose **Create record**.
2. Use the wizard method to create the record. If the console presents you the **Quick create** method, choose **Switch to wizard**.
3. Choose **Simple Routing** for **Routing policy** and then choose **Next**.
4. In the **Configure records** page, choose **Define simple record**.
5. In the **Define simple record** page:
   - For **Record name**, enter `iot:Data-ATS` endpoint or `iot:CredentialProvider` endpoint. This must be the same as the private hosted zone name.
   - For **Record type**, if you want only IPv4 support, keep the value as `A - Routes traffic to an IPv4 address and some AWS resources`. If you want only IPv6 support, keep the value as `AAAA - Routes traffic to an IPv6 address and some AWS resources`. If you want dual-stack support (both IPv4 and IPv6), create two records (`A` and `AAAA` in the hosted zone with the same **Record name** and **Value/Route traffic to**.
   - For **Value/Route traffic to**, choose **Alias to VPC endpoint**. Then choose your **Region** and then choose the endpoint that you created previously, as described in [???]() from the list of endpoints displayed.

6. Choose **Define simple record** to create your record.

Controlling Access to AWS IoT Core over VPC endpoints

You can restrict device access to AWS IoT Core to be allowed only through VPC endpoint by using VPC condition context keys. AWS IoT Core supports the following VPC related context keys:

- SourceVpc
- SourceVpce
- VPCSourceIp

Note

AWS IoT Core doesn't support Endpoints policies for VPC endpoints.

For example, the following policy grants permission to connect to AWS IoT Core using a client ID that matches the thing name, and to publish to any topic prefixed by the thing name, conditional on the device connecting to a VPC endpoint with a particular VPC Endpoint ID. This policy would deny connection attempts to your public IoT data endpoint.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:SourceVpce": "vpce-1a2b3c4d"
                }
        }
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingName}/*"
            ]
        }
    ]
}
```

Limitations

VPC endpoints are currently supported for AWS IoT Core control plane endpoints, AWS IoT Core data endpoints, and AWS IoT Core credential provider endpoints. VPC endpoints are only supported for Federal Information Processing Standard (FIPS) endpoints when using the AWS IoT Core control plane.

Limitations of IoT control plane VPC endpoints

This section covers the limitations of IoT control plane VPC endpoints.

- VPC endpoints will serve ATS certificates only.
- Custom domains are not supported for control plane endpoints.
- For information regarding FIPS security policies, see FIPS security policies.

Limitations of IoT data VPC endpoints

This section covers the limitations of IoT data VPC endpoints.

- MQTT keep alive periods are limited to 230 seconds. Keep alive periods longer than that will be automatically reduced to 230 seconds.
- Each VPC endpoint supports 100,000 total concurrent connected devices. If you require more connections see Scaling VPC endpoints with AWS IoT Core .

- VPC endpoints will serve ATS certificates only, except for custom domains.
- VPC endpoint policies are not supported.
- For VPC endpoints that are created for the AWS IoT Core data plane, AWS IoT Core doesn't support using zonal or regional public DNS records.

Limitations of credential provider endpoints

This section covers the limitations of credential provider VPC endpoints.

- VPC endpoints will serve ATS certificates only.
- VPC endpoint policies are not supported.
- Custom domains are not supported for credential provider endpoints.
- For VPC endpoints that are created for the AWS IoT Core credential provider, AWS IoT Core doesn't support using zonal or regional public DNS records.

Scaling VPC endpoints with AWS IoT Core

AWS IoT Core Interface VPC endpoints are limited to 100,000 connected devices over a single interface endpoint. If your use case calls for more concurrent connections to the broker, then we recommend using multiple VPC endpoints and manually routing your devices across your interface endpoints. When creating private DNS records to route traffic to your VPC endpoints, make sure to create as many weighted records as you have VPC endpoints to distribute traffic across your multiple endpoints.

Using custom domains with VPC endpoints

If you want to use custom domains with VPC endpoints, you must create your custom domain name records in a private hosted zone and create routing records in Route53. For more information, see Creating A private hosted zone.

> **Note**
>
> Custom domains are only supported for AWS IoT Core data endpoints.

Availability of VPC endpoints for AWS IoT Core

AWS IoT Core Interface VPC endpoints are available in all AWS IoT Core supported regions. AWS IoT Core Interface VPC endpoints for AWS IoT Core credential provider are not supported in China Region and AWS GovCloud (US) Regions.

Using AWS IoT Device Management secure tunneling with interface VPC endpoints

AWS IoT Device Management secure tunneling supports interface VPC endpoints. You can use VPC endpoints to keep traffic between your VPC and AWS IoT Secure Tunneling within the AWS network, without requiring an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection.

Interface VPC endpoints are powered by AWS PrivateLink, a technology that enables you to privately access services by using private IP addresses. For more information, see Access an AWS service using an interface VPC endpoint in the AWS PrivateLink Guide.

Contents

Prerequisites

Before you create VPC endpoints for AWS IoT Secure Tunneling, verify that you have the following:

- An AWS account with the necessary permissions to create VPC endpoints.
- A VPC in your AWS account.
- Understanding of AWS IoT Device Management secure tunneling concepts.
- Familiarity with VPC endpoint policies and AWS Identity and Access Management (IAM)

Receiving tunnel notifications through VPC endpoints

To receive tunnel notifications through a VPC endpoint, your devices can connect to the AWS IoT Core data plane through a VPC endpoint and subscribe to the secure tunneling reserved MQTT topic.

For instructions on how to create and configure a VPC endpoint in the AWS IoT Core data plane, see Using AWS IoT Core with interface VPC endpoints in the AWS IoT Developer Guide.

Creating VPC endpoints for secure tunneling

You can create VPC endpoints for both secure tunneling control plane and proxy server.

To create a VPC endpoint for secure tunneling

1. Follow the steps in Creating an interface endpoint in the Amazon VPC Developer Guide
2. For **Service name**, choose one of the following options based on your endpoint type:

   Control plane

   - Standard: `com.amazonaws.<region>.iot.tunneling.api`
   - FIPS (available in FIPS regions): `com.amazonaws.<region>.iot-fips.tunneling.api`

   Proxy server

   - Standard: `com.amazonaws.<region>.iot.tunneling.data`
   - FIPS (available in FIPS regions): `com.amazonaws.<region>.iot-fips.tunneling.data`

   Replace *`<region>`* with your AWS Region. For example, `us-east-1`.
3. Complete the remaining steps in the VPC endpoint creation process according to your network requirements.

Configuring VPC endpoint policies on Proxy Server

In addition to client access token-based authorization that is used to authorize connections to tunnels, you can use VPC endpoint policies to further restrict how devices can use a VPC endpoint to connect to the Secure Tunneling Proxy Server. VPC endpoint policies follow an IAM-like syntax and are configured on the VPC endpoint itself.

Note that the only supported IAM action for proxy server VPC endpoint policies is `iot:ConnectToTunnel`.

Below are examples of different VPC endpoint policies.

Proxy server VPC endpoint policy examples

The following examples show Proxy Server VPC endpoint policy configurations for common use cases.

Example- Default policy

This policy allows devices within your VPC to connect to any tunnel in the same AWS Region where the endpoint is created, across any AWS account.

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

Example- Restrict access to specific AWS accounts

This policy allows the VPC endpoint to connect only to tunnels in specific AWS accounts.

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": [
                "arn:aws:iot:us-east-1:111122223333:tunnel/*",
                "arn:aws:iot:us-east-1:444455556666:tunnel/*"
            ]
        }
    ]
```

```
}
```

Example- Restrict connections by tunnel endpoint

You can restrict VPC endpoint access to only allow devices to connect to the source or destination end of a tunnel.

Source only:

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iot:ClientMode": "source"
                }
            }
        }
    ]
}
```

Destination only:

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iot:ClientMode": "destination"
                }
            }
        }
    ]
}
```

Example- Restrict access based on resource tags

This policy allows the VPC endpoint to connect only to tunnels that are tagged with a specific key-value pair.

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Environment": "Production"
                }
            }
        }
    ]
}
```

Example- Combined policy conditions

This policy demonstrates combining multiple policy elements. It allows connections to any tunnel in a specific AWS account, but only if the tunnel is tagged with `AllowConnectionsThroughPrivateLink` set to `true` and the client is not connecting to the destination end of the tunnel.

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": [
                "arn:aws:iot:us-east-1:111122223333:tunnel/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/AllowConnectionsThroughPrivateLink": "true"
                }
            }
```

```
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "iot:ConnectToTunnel",
            "Resource": [
                "arn:aws:iot:us-east-1:111122223333:tunnel/*"
            ],
            "Condition": {
                "StringEquals": {
                    "iot:ClientMode": "destination"
                }
            }
        }
    ]
}
```

Next steps

After you create and configure your VPC endpoints for AWS IoT Secure Tunneling, consider the following:

- Test your VPC endpoint configuration by connecting devices through the endpoint.
- Monitor VPC endpoint usage through Amazon CloudWatch metrics.
- Review and update your VPC endpoint policies as needed for your security requirements.

For more information about AWS IoT Device Management secure tunneling, see AWS IoT Secure Tunneling.

Infrastructure security in AWS IoT

As a collection of managed services, AWS IoT is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access AWS IoT through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems, such as Java 7 and later, support these modes. For more information, see Transport security in AWS IoT Core.

Requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Security monitoring of production fleets or devices with AWS IoT Core

IoT fleets can consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make fleet setup complex and prone to errors. And because devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. These factors make IoT fleets an attractive target for hackers and make it difficult to secure your device fleet on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. You can use AWS IoT Device Defender to analyze, audit, and monitor connected devices to detect abnormal behavior, and mitigate security risks. AWS IoT Device Defender can audit device fleets to ensure they adhere to security best practices and detect abnormal behavior on devices. This makes it possible to enforce consistent security policies across your AWS IoT device fleet and respond quickly when devices are compromised. For more information, see AWS IoT Device Defender.

AWS IoT Device Advisor pushes updates and patches your fleet as needed. AWS IoT Device Advisor updates test cases automatically. The test cases that you select are always with latest version. For more information, see Device Advisor.

Security best practices in AWS IoT Core

This section contains information about security best practices for AWS IoT Core. For information about security rules for Industrial IoT solutions, see Ten security golden rules for Industrial IoT solutions.

Protecting MQTT connections in AWS IoT

AWS IoT Core is a managed cloud service that makes it possible for connected devices to interact with cloud applications and other devices easily and securely. AWS IoT Core supports HTTP, WebSocket, and MQTT, a lightweight communication protocol specifically designed to tolerate intermittent connections. If you are connecting to AWS IoT using MQTT, each of your connections must be associated with an identifier known as a client ID. MQTT client IDs uniquely identify MQTT connections. If a new connection is established using a client ID that is already claimed for another connection, the AWS IoT message broker drops the old connection to allow the new connection. Client IDs must be unique within each AWS account and each AWS Region. This means that you don't need to enforce global uniqueness of client IDs outside of your AWS account or across Regions within your AWS account.

The impact and severity of dropping MQTT connections on your device fleet depends on many factors. These include:

- Your use case (for example, the data your devices send to AWS IoT, how much data, and the frequency that the data is sent).
- Your MQTT client configuration (for example, auto reconnect settings, associated back-off timings, and use of MQTT persistent sessions).
- Device resource constraints.
- The root cause of the disconnections, its aggressiveness, and persistence.

To avoid client ID conflicts and their potential negative impacts, make sure that each device or mobile application has an AWS IoT or IAM policy that restricts which client IDs can be used for MQTT connections to the AWS IoT message broker. For example, you can use an IAM policy to prevent a device from unintentionally closing another device's connection by using a client ID that is already in use. For more information, see Authorization.

All devices in your fleet must have credentials with privileges that authorize intended actions only, which include (but not limited to) AWS IoT MQTT actions such as publishing messages or subscribing to topics with specific scope and context. The specific permission policies can vary for your use cases. Identify the permission policies that best meet your business and security requirements.

To simplify creation and management of permission policies, you can use AWS IoT Core policy variables and IAM policy variables. Policy variables can be placed in a policy and when the policy is evaluated, the variables are replaced by values that come from the device's request. Using policy variables, you can create a single policy for granting permissions to multiple devices. You can identify the relevant policy variables for your use case based on your AWS IoT account configuration, authentication mechanism, and network protocol used in connecting to AWS IoT message broker. However, to write the best permission policies, consider the specifics of your use case and your threat model.

For example, if you registered your devices in the AWS IoT registry, you can use thing policy variables in AWS IoT policies to grant or deny permissions based on thing properties like thing names, thing types, and thing attribute values. The thing name is obtained from the client ID in the MQTT connect message sent when a thing connects to AWS IoT. The thing policy variables are replaced when a thing connects to AWS IoT over MQTT using TLS mutual authentication or MQTT over the WebSocket protocol using authenticated Amazon Cognito identities. You can use the AttachThingPrincipal API to attach certificates and authenticated Amazon Cognito identities to a thing. `iot:Connection.Thing.ThingName` is a useful thing policy variable to enforce client ID restrictions. The following example AWS IoT policy requires a registered thing's name to be used as the client ID for MQTT connections to the AWS IoT message broker:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": "iot:Connect",
   "Resource": [
    "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
   ]
  }
 ]
}
```

If you want to identify ongoing client ID conflicts, you can enable and use CloudWatch Logs for AWS IoT. For every MQTT connection that the AWS IoT message broker disconnects due to client ID conflicts, a log record similar to the following is generated:

```
{
    "timestamp": "2019-04-28 22:05:30.105",
    "logLevel": "ERROR",
    "traceId": "02a04a93-0b3a-b608-a27c-1ae8ebdb032a",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "Disconnect",
    "protocol": "MQTT",
```

```
    "clientId": "clientId01",
    "principalId": "1670fcf6de55adc1930169142405c4a2493d9eb5487127cd0091ca0193a3d3f6",
    "sourceIp": "203.0.113.1",
    "sourcePort": 21335,
    "reason": "DUPLICATE_CLIENT_ID",
    "details": "A new connection was established with the same client ID"
}
```

You can use a CloudWatch Logs filter such as `{$.reason= "DUPLICATE_CLIENT_ID" }` to search for instances of client ID conflicts or to set up CloudWatch metric filters and corresponding CloudWatch alarms for continuous monitoring and reporting.

You can use AWS IoT Device Defender to identify overly permissive AWS IoT and IAM policies. AWS IoT Device Defender also provides an audit check that notifies you if multiple devices in your fleet are connecting to the AWS IoT message broker using the same client ID.

You can use AWS IoT Device Advisor to validate that your devices can reliably connect to AWS IoT Core and follow security best practices.

See also

- AWS IoT Core
- AWS IoT's Security Features
- AWS IoT Core policy variables
- IAM Policy Variables
- Amazon Cognito Identity
- AWS IoT Device Defender
- CloudWatch Logs for AWS IoT

Keep your device's clock in sync

It's important to have an accurate time on your device. X.509 certificates have an expiry date and time. The clock on your device is used to verify that a server certificate is still valid. If you're building commercial IoT devices, remember that your products may be stored for extended periods before being sold. Real-time clocks can drift during this time and batteries can get discharged, so setting time in the factory is not sufficient.

For most systems, this means that the device's software must include a network time protocol (NTP) client. The device should wait until it synchronizes with an NTP server before it tries to connect to AWS IoT Core. If this isn't possible, the system should provide a way for a user to set the device's time so that subsequent connections succeed.

After the device synchronizes with an NTP server, it can open a connection with AWS IoT Core. How much clock skew that is allowed depends on what you're trying to do with the connection.

Validate the server certificate

The first thing a device does to interact with AWS IoT is to open a secure connection. When you connect your device to AWS IoT, ensure that you're talking to AWS IoT and not another server impersonating AWS IoT. Each of the AWS IoT servers is provisioned with a certificate issued for the `iot.amazonaws.com` domain. This certificate was issued to AWS IoT by a trusted certificate authority that verified our identity and ownership of the domain.

One of the first things AWS IoT Core does when a device connects is send the device a server certificate. Devices can verify that they were expecting to connect to `iot.amazonaws.com` and that the server on the end of that connection possesses a certificate from a trusted authority for that domain.

TLS certificates are in X.509 format and include a variety of information such as the organization's name, location, domain name, and a validity period. The validity period is specified as a pair of time values called `notBefore` and `notAfter`. Services like AWS IoT Core use limited validity periods (for example, one year) for their server certificates and begin serving new ones before the old ones expire.

Use a single identity per device

Use a single identity per client. Devices generally use X.509 client certificates. Web and mobile applications use Amazon Cognito Identity. This enables you to apply fine-grained permissions to your devices.

For example, you have an application that consists of a mobile phone device that receives status updates from two different smart home objects – a light bulb and a thermostat. The light bulb sends the status of its battery level, and a thermostat sends messages that report the temperature.

AWS IoT authenticates devices individually and treats each connection individually. You can apply fine-grained access controls using authorization policies. You can define a policy for the thermostat that allows it to publish to a topic space. You can define a separate policy for the light bulb that allows it to publish to a different topic space. Finally, you can define a policy for the mobile app that only allows it to connect and subscribe to the topics for the thermostat and the light bulb to receive messages from these devices.

Apply the principle of least privilege and scope down the permissions per device as much as possible. All devices or users should have an AWS IoT policy in AWS IoT that only allows it to connect with a known client ID, and to publish and subscribe to an identified and fixed set of topics.

Use a second AWS Region as backup

Consider storing a copy of your data in a second AWS Region as a backup. Note that the AWS solution named Disaster Recovery for AWS IoT is no longer available. While the associated GitHub library remains accessible, AWS deprecated it in July 2023 and no longer provides maintenance or support for it. To implement your own solutions or to explore additional support options, visit Contact AWS. If there is an AWS Technical Account Manager associated with your account, reach out to them for help.

Use just in time provisioning

Manually creating and provisioning each device can be time consuming. AWS IoT provides a way to define a template to provision devices when they first connect to AWS IoT. For more information, see Just-in-time provisioning.

Permissions to run AWS IoT Device Advisor tests

The following policy template shows the minimum permissions and IAM entity required to run AWS IoT Device Advisor test cases. You will need to replace *your-device-role-arn* with the device role Amazon Resource Name (ARN) that you created under the prerequisites.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iot:us-east-1:123456789012:thinggroup/your-thing-
group",
            "Condition": {
```

```
                "StringEquals": {
                    "iam:PassedToService": "iotdeviceadvisor.amazonaws.com"
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke*",
                "iam:ListRoles",
                "iot:Connect",
                "iot:CreateJob",
                "iot:DeleteJob",
                "iot:DescribeCertificate",
                "iot:DescribeEndpoint",
                "iotjobsdata:DescribeJobExecution",
                "iot:DescribeJob",
                "iot:DescribeThing",
                "iotjobsdata:GetPendingJobExecutions",
                "iot:GetPolicy",
                "iot:ListAttachedPolicies",
                "iot:ListCertificates",
                "iot:ListPrincipalPolicies",
                "iot:ListThingPrincipals",
                "iot:ListThings",
                "iot:Publish",
                "iotjobsdata:StartNextPendingJobExecution",
                "iotjobsdata:UpdateJobExecution",
                "iot:UpdateThingShadow",
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:PutRetentionPolicy"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": "iotdeviceadvisor:*",
            "Resource": "*"
```

```
        }
    ]
}
```

Cross-service confused deputy prevention for Device Advisor

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Device Advisor gives another service to the resource. If you use both global condition context keys, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

The value of `aws:SourceArn` must be the ARN of your suite definition resource. The suite definition resource refers to the test suite you created with Device Advisor.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcards (*) for the unknown portions of the ARN. For example, `arn:aws:iotdeviceadvisor:*:account-id:suitedefinition/*`

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Device Advisor to prevent the confused deputy problem.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Sid": "ConfusedDeputyPreventionExamplePolicy",
        "Effect": "Allow",
```

```
        "Principal": {
            "Service": "iotdeviceadvisor.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "ArnLike": {
                "aws:SourceArn": "arn:aws:iotdeviceadvisor:us-
east-1:123456789012:suitedefinition/ygp6rxa3tzvn"
        },
            "StringEquals": {
                "aws:SourceAccount": "123456789012"
        }
        }
    }
}
```

AWS training and certification

Take the following course to learn about key concepts for AWS IoT security: AWS IoT Security Primer.

Monitoring AWS IoT

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT and your AWS solutions.

We strongly encourage you to collect monitoring data from all parts of your AWS solution to make it easier to debug a multi-point failure, if one occurs. Start by creating a monitoring plan that answers the following questions. If you're not sure how to answer these, you can still continue to enable logging and establish your performance baselines.

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Your next step is to enable logging and establish a baseline of normal AWS IoT performance in your environment by measuring performance at various times and under different load conditions. As you monitor AWS IoT, keep historical monitoring data so that you can compare it with current performance data. This will help you identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish your baseline performance for AWS IoT, you should monitor these metrics to start. You can always monitor more metrics later.

- `PublishIn.Success`
- `PublishOut.Success`
- `Subscribe.Success`
- `Ping.Success`
- `Connect.Success`
- `GetThingShadow.Accepted`
- `UpdateThingShadow.Accepted`
- `DeleteThingShadow.Accepted`

- `RulesExecuted`

The topics in this section can help you start logging and monitoring AWS IoT.

Topics

Configure AWS IoT logging

You must enable logging by using the AWS IoT console, CLI, or API before you can monitor and log AWS IoT activity. You can configure logging for AWS IoT at three levels: account-level, event-level or resource-specific level. Event-level and resource-specific logging are available exclusively with V2 logging. Customers using V1 logging must perform a migration to V2 to access these features. See details.

When considering how to configure your AWS IoT logging, the account-level logging configuration determines how AWS IoT activity will be logged unless specified otherwise. Starting out, you might want to obtain detailed logs with a default log level of INFO or DEBUG. After reviewing the initial logs, you can change the default log level to a less verbose level such as WARN or ERROR at account or event level and set a more verbose resource-specific log level on resources that might need more attention. Log levels can be changed whenever you want.

This topic covers cloud-side logging in AWS IoT. For information on device-side logging and monitoring, see Upload device-side logs to CloudWatch.

For information on logging and monitoring AWS IoT Greengrass, see Logging and monitoring in AWS IoT Greengrass.

Configuring V2 logging in AWS IoT

Determining your logging version

The GetV2LoggingOptions API returns a NotConfiguredException if V2 logging is not enabled. This error occurs when either V1 logging is in use, or no logging has been configured.

Understanding V2 logging features

V2 logging provides two key capabilities: event-level logging and resource-specific logging. Event-level logging enables targeted logging configuration with customizable log levels and CloudWatch log group destinations. Resource-specific logging allows you to filter logs by thing group, source IP, client ID, or principal ID. Together, these features provide granular control and comprehensive visibility into IoT operations, improving log searchability and reducing costs by eliminating unnecessary logging activity.

Migrating from V1 to V2

You can migrate to V2 logging using the SetV2LoggingOptions API through either the AWS CLI or SDK. After migration, AWS IoT automatically routes all logs to the CloudWatch log group 'AWSIotLogsV2'. Important: If your downstream applications or resources consume information from ‘AWSIotLogs’, update them to use the corresponding log group path.

Configure logging role and policy

Before you can enable logging in AWS IoT, you must create an IAM role and a policy that gives AWS IoT permission to write AWS IoT log activities to CloudWatch log groups on your behalf. You can also generate an IAM role with the policies needed in the Logs section of the AWS IoT console.

> **Note**
>
> Before enabling AWS IoT logging, ensure you understand the CloudWatch Logs access permissions. Users with access to CloudWatch Logs can see debugging information from your devices. For more information, see Authentication and Access Control for Amazon CloudWatch Logs.
> If you expect high traffic patterns in AWS IoT Core due to load testing, consider disabling IoT logging to prevent throttling. If high traffic is detected, our service may disable logging in your account.

The following shows how to create a logging role and policy for AWS IoT Core resources.

Create a logging role

To create a logging role, open the [Roles hub of the IAM console]() and choose **Create role**.

1. Under **Select trusted entity**, choose **AWS Service**. Then choose **IoT** under **Use case**. If you don't see **IoT**, enter and search **IoT** from the **Use cases for other AWS services:** drop-down menu. Select **Next**.
2. On the **Add permissions** page, you will see the policies that are automatically attached to the service role. Choose **Next**.
3. On the **Name, review, and create** page, enter a **Role name** and **Role description** for the role, then choose **Create role**.

Logging role policy

The following policy documents provide the role policy and trust policy that allow AWS IoT to submit log entries to CloudWatch on your behalf. If you configure event-level logging with a custom CloudWatch log group, you must update the role policy to include the custom resource ARN.

If you also allowed AWS IoT Core for LoRaWAN to submit log entries, you'll see a policy document created for you that logs both activities.

> **Note**
>
> These documents were created for you when you created the logging role. The documents have variables, `${partition},` `${region}`, and `${accountId}`, that you must replace with your values.
>
> - Replace partition with the partition of the region.
> - Replace region with the AWS Region that you use. Make sure that you use the same AWS Region that you used to configure the AWS CLI on your device.
> - Replace account-id with your AWS account ID.

Role policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:PutMetricFilter",
                "logs:PutRetentionPolicy",
                "iot:GetLoggingOptions",
                "iot:SetLoggingOptions",
                "iot:SetV2LoggingOptions",
                "iot:GetV2LoggingOptions",
                "iot:SetV2LoggingLevel",
                "iot:ListV2LoggingLevels",
                "iot:DeleteV2LoggingLevel"
            ],
            "Resource": [
                "arn:aws:logs:us-east-1:123456789012:log-group:AWSIotLogsV2:*"
            ]
        }
    ]
}
```

Trust policy to log only AWS IoT Core activity:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
        },
            "Action": "sts:AssumeRole"
        }
    ]
```

```
}
```

AWS IoT Logging may fail to publish logging to CloudWatch Logs due to insufficient IAM role permissions. When this occurs, check CloudWatch logging metrics to investigate and troubleshoot the failures.

Configure logging in the AWS IoT (console)

This section describes how to configure AWS IoT logging using the AWS IoT console. You can set-up account-level, event-level and resource-specific logging.

To configure AWS IoT logging:

1. Sign in to the AWS IoT console. For more information, see Open the AWS IoT console.
2. In the left navigation pane, choose **Logs** (previously a section under Settings).
3. Configure account-level logging: account level logging applies to all your AWS IoT fleet (devices or endpoints), unless overridden by event-level or resource-specific settings.
   a. Under Account-level logging, select **Manage account-level logging** to make updates.
   b. Select the "Enable logging" checkbox to start sending logs to CloudWatch. When “Enable logging” is unchecked, AWS IoT will not send any logs to CloudWatch log groups, regardless of event-level or resource-level logging configurations.
   c. Under **IAM log role**, select an existing role from the dropdown list. You can **View role details** to inspect role permissions. Alternatively, select **Create new role** to set up a new IAM role. The logging role provides policies that allow AWS IoT to submit log entries to CloudWatch on your behalf. If you configure event-level logging with a custom CloudWatch log group, you must update the role policy to include the ARN for this log group.
   d. Choose the **Default Log level** that corresponds to the level of detail of the log entries that you want to appear in the CloudWatch Logs. Note: Log level “DEBUG” provides the most detail but increases CloudWatch costs. CloudWatch log group destinations cannot be configured at the account level. However, you can specify custom log groups for individual event types as described in the following section.
   e. Choose **Update logging** to save your changes.
4. Event-level logging allows you to selectively capture logs for relevant events and direct them to dedicated CloudWatch log groups. This gives you the flexibility to organize logs by use cases

for better discoverability, share them with different audiences, and reduce CloudWatch costs by enabling logs and setting log levels based on event criticality.

Configure **event-level logging**: event-level logging captures specific AWS IoT events, such as client authentication attempts. These settings override Account-Level Logging.

a. Under **Event-level logging** section, select **Manage event-level logging** to make updates.
b. By default, event types inherit the account-level logging configuration. Note: When resource-specific logging is configured, it overrides account and event-level settings.
c. To modify settings for individual events, click the value in the corresponding event row. You can adjust both the log level and the CloudWatch log group destination. When specifying a custom CloudWatch log group destination, you must verify that the IAM role policy includes permissions for the new log group. Failure to update the role policy will prevent AWS IoT from writing logs to the custom log group. After making your selection, click the check mark to confirm your choice. The **'Is Modified'** column will display 'Yes' to indicate pending changes.
d. Click on **Update Logging** to apply your changes or choose **Cancel** to discard.

5. Configure Resource-specific overrides: Resource-specific overrides apply a logging setting to selected resources. A resource can be a thing group, source IP, client ID, or principal ID. Resource-specific logging configuration overrides both account-level and event-level settings. When enabled, it generates logs for all event types at the configured logging level for the specified resources. For example, you can set debug-level logging for a specific Thing while keeping info-level logging for all other Things.

   a. Select **Add resource-specific overrides** in the Resource-specific overrides section.
   b. Choose a log target: Thing group, Source IP, Client ID or Principal ID.
   c. Enter the corresponding log target value for your selected target type.
   d. Select the desired log level from the dropdown menu in the Resource-specific log level section.
   e. Click **Submit** to add the override or **Cancel** to discard changes.
   f. To modify an existing resource-specific override, select the checkbox next to the resource and click “Remove” to delete the override or “Edit” to modify.

After you've enabled logging, visit [Viewing AWS IoT logs in the CloudWatch console](#) to learn more about viewing the log entries.

Configure Account and Event-level logging in AWS IoT (CLI)

This section describes how to configure global logging for AWS IoT by using the CLI.

You can optionally configure Event-level logging. Event-level logging captures logging information at the event level, such as authentication and authorization or certificate creation events. You can customize both the log level and the CloudWatch log group destinations at the event level. Event-level logging operates at a more targeted level compared with account-level logging and therefore overrides the account-level logging settings. This hierarchical approach allows you to maintain different logging strategies for different types of events based on their operational importance and cost considerations.

> **Note**
>
> You need the Amazon Resource Name (ARN) of the role that you want to use. If you need to create a role to use for logging, see Create a logging role before continuing. When specifying a custom CloudWatch log group for any event type, ensure your logging role has the required permissions for the target log group.
> The principal used to call the API must have Passing role permissions for your logging role.

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

To use the CLI to configure default logging for AWS IoT

1. Use the **set-v2-logging-options** command to set the logging options for your account.

   ```
   aws iot set-v2-logging-options \
       --event-configurations event-configuration-list \
       --role-arn logging-role-arn \
       --default-log-level log-level
   ```

   where:

   --role-arn

   The role ARN that grants AWS IoT permission to write to your logs in CloudWatch Logs. Role-arn configuration is required for initial setup.

--default-log-level

The [log level](#) to use. Valid values are: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `DISABLED`. Default-log-level configuration is required for initial setup.

--no-disable-all-logs

An optional parameter that enables all AWS IoT logging. Use this parameter to enable logging when it is currently disabled.

--disable-all-logs

An optional parameter that disables all AWS IoT logging. Use this parameter to disable logging when it is currently enabled.

--event-configurations

This parameter is optional and allows you to customize logging settings for individual event types:

- eventType: The type of event that overrides the account-level setting
- logLevel: Override the account-level setting with DEBUG, INFO, ERROR, WARN, or DISABLED
- logDestination: Specify a custom CloudWatch log group for log delivery

You can configure logging level and log destination independently for each event type. If not specified, events will inherit the account-level settings

```
aws iot set-v2-logging-options \
    --event-configurations "[{\"eventType\":\"Publish-In\",\"logLevel\":\"INFO\",
\"logDestination\":\"examplePublishInLogGroup\"}]"
```

2. Use the **get-v2-logging-options** command to get your current logging options.

```
aws iot get-v2-logging-options \
    --verbose
```

where:

--verbose

An optional parameter that enables you to retrieve all eventTypes and their configurations.

After you've enabled logging, visit Viewing AWS IoT logs in the CloudWatch console to learn more about viewing the log entries.

> **Note**
>
> AWS IoT continues to support older commands (**set-logging-options** and **get-logging-options**) to set and get global logging on your account. Be aware that when these commands are used, the resulting logs contain plain-text, rather than JSON payloads and logging latency is generally higher. No further improvements will be made to the implementation of these older commands. We recommend that you use the "v2" versions to configure your logging options and, when possible, change legacy applications that use the older versions.

Configure Resource-specific overrides in AWS IoT (CLI)

This section describes how to configure Resource-specific overrides for AWS IoT by using the CLI. Resource-specific overrides allows you to specify a logging level for a specific resource identified by Thing Group, Client ID, Source IP, or Principal ID. When resource-specific logging is enabled, it overrides both account-level and event-level settings. All event types will generate logs for the specified resource at the configured logging level, even if those events are disabled in the event-level configurations.

Thing groups can contain other thing groups to create a hierarchical relationship. This procedure describes how to configure the logging of a single thing group. You can apply this procedure to the parent thing group in a hierarchy to configure the logging for all thing groups in the hierarchy. You can also apply this procedure to a child thing group to override the logging configuration of its parent.

A thing can be a member of a thing group. This membership allows the thing to inherit configurations, policies, and settings applied to the thing group. Thing groups are used to manage and apply settings to multiple things collectively, rather than dealing with each thing individually. When your client ID matches the thing name, AWS IoT Core will automatically associate the client session with the corresponding thing resource. This allows the client session to inherit the configurations and settings applied to the thing groups to which the thing belongs, including the logging levels. If your client ID doesn't match the thing name, you can enable the exclusive thing attachment to establish the association. For more information, see ??? .

In addition to thing groups, you can also log targets such as a device's client ID, source IP, and principal ID.

> **Note**
>
> You need the Amazon Resource Name (ARN) of the role you want to use. If you need to create a role to use for logging, see Create a logging role before continuing.
> The principal used to call the API must have Passing role permissions for your logging role.

You can also perform this procedure with the API by using the methods in the AWS API that correspond to the CLI commands shown here.

To use the CLI to configure Resource-specific overrides for AWS IoT

1. Enable account-level logging before configuring resource-specific logging using the following command: aws iot set-v2-logging-options command
2. Use the **set-v2-logging-level** command to configure Resource-specific overrides. See the following example for thing group configuration:

```
aws iot set-v2-logging-level \
              --log-target targetType=THING_GROUP,targetName=thing_group_name \
              --log-level log_level
```

--log-target

The type and name of the resource for which you are configuring logging. The `targetType` value must be one of: `THING_GROUP` | `CLIENT_ID` | `SOURCE_IP` | `PRINCIPAL_ID`. The log-target parameter value can be text, as shown in the preceding command example, or a JSON string, such as the following example.

```
aws iot set-v2-logging-level \
              --log-target '{"targetType": "THING_GROUP","targetName":
 "thing_group_name"}' \
              --log-level log_level
```

--log-level

The logging level used when generating logs for the specified resource. Valid values are: **DEBUG**, **INFO**, **ERROR**, **WARN**, and **DISABLED**.

3. Use the [list-v2-logging-levels] command to list the currently configured logging levels.

```
aws iot list-v2-logging-levels
```

4. Use the [delete-v2-logging-level] command to delete a resource-specific logging level, such as the following examples.

```
aws iot delete-v2-logging-level \
              --target-type "THING_GROUP" \
              --target-name "thing_group_name"
```

```
aws iot delete-v2-logging-level \
              --target-type=CLIENT_ID
              --target-name=ClientId1
```

--target-type

The `target-type` value must be one of: `THING_GROUP` | `CLIENT_ID` | `SOURCE_IP` | `PRINCIPAL_ID`.

--target-name

The name of the thing group for which to remove the logging level.

Log levels

These log levels determine the events that are logged and apply to default and resource-specific log levels.

ERROR

Any error that causes an operation to fail.

Example: failed to authenticate device due to expired certificate.

Logs include ERROR information only.

WARN

Anything that can potentially cause inconsistencies in the system, but might not cause the operation to fail.

Example: approaching message rate limit.

Logs include ERROR and WARN information.

INFO

High-level information about the flow of things.

Example: a client has successfully subscribed to an MQTT topic.

Logs include INFO, ERROR, and WARN information.

DEBUG

Information that might be helpful when debugging a problem.

Example: IoT Rules Engine detected a message published to the rules topic “rule/test” and successfully started executing. The rule is configured with RepublishAction.

Logs include DEBUG, INFO, ERROR, and WARN information.

DISABLED

All logging is disabled.

Monitor AWS IoT alarms and metrics using Amazon CloudWatch

You can monitor AWS IoT using CloudWatch, which collects and processes raw data from AWS IoT into readable, near real-time metrics. These statistics are recorded for a period of two weeks, so that you can access historical information and gain a better perspective on how your web application or service is performing. By default, AWS IoT metric data is sent automatically to CloudWatch in one minute intervals. For more information, see What Are Amazon CloudWatch, Amazon CloudWatch Events, and Amazon CloudWatch Logs? in the *Amazon CloudWatch User Guide*.

Using AWS IoT metrics

The metrics reported by AWS IoT provide information that you can analyze in different ways. The following use cases are based on a scenario where you have ten things that connect to the internet once a day. Each day:

- Ten things connect to AWS IoT at roughly the same time.
- Each thing subscribes to a topic filter, and then waits for an hour before disconnecting. During this period, things communicate with one another and learn more about the state of the world.
- Each thing publishes some perception it has based on its newly found data using `UpdateThingShadow`.
- Each thing disconnects from AWS IoT.

To help you get started, these topics explore some of the questions that you might have.

- How can I be notified if my things do not connect successfully each day?
- How can I be notified if my things are not publishing data each day?
- How can I be notified if my thing's shadow updates are being rejected each day?
- How can I create a CloudWatch alarm for Jobs?

More about CloudWatch alarms and metrics

- Creating CloudWatch alarms to monitor AWS IoT
- AWS IoT metrics and dimensions

Creating CloudWatch alarms to monitor AWS IoT

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period you specify. When the value of the metric exceeds a given threshold over a number of time periods, one or more actions are performed. The action can be a notification sent to an Amazon SNS topic or Auto Scaling policy. Alarms trigger actions for sustained state changes only. CloudWatch alarms do not trigger actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

The following topics describe some examples of using CloudWatch alarms.

- How can I be notified if my things do not connect successfully each day?
- How can I be notified if my things are not publishing data each day?
- How can I be notified if my thing's shadow updates are being rejected each day?
- How can I create a CloudWatch alarm for jobs?

You can see all the metrics that CloudWatch alarms can monitor at AWS IoT metrics and dimensions .

How can I be notified if my things do not connect successfully each day?

1. Create an Amazon SNS topic named `things-not-connecting-successfully`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as *sns-topic-arn* .

   For more information on how to create an Amazon SNS notification, see Getting Started with Amazon SNS.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name ConnectSuccessAlarm \
    --alarm-description "Alarm when my Things don't connect successfully" \
    --namespace AWS/IoT \
    --metric-name Connect.Success \
    --dimensions Name=Protocol,Value=MQTT \
    --statistic Sum \
    --threshold 10 \
    --comparison-operator LessThanThreshold \
    --period 86400 \
    --evaluation-periods 1 \
    --alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name ConnectSuccessAlarm --state-reason
 "initializing" --state-value ALARM
```

4. Verify that the alarm appears in your [CloudWatch console](https://console.aws.amazon.com/cloudwatch/).

How can I be notified if my things are not publishing data each day?

1. Create an Amazon SNS topic named `things-not-publishing-data`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as *sns-topic-arn* .

   For more information on how to create an Amazon SNS notification, see [Getting Started with Amazon SNS](https://docs.aws.amazon.com/sns/latest/dg/sns-getting-started.html).

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name PublishInSuccessAlarm\
    --alarm-description "Alarm when my Things don't publish their data \
    --namespace AWS/IoT \
    --metric-name PublishIn.Success \
    --dimensions Name=Protocol,Value=MQTT \
    --statistic Sum \
    --threshold 10 \
    --comparison-operator LessThanThreshold \
    --period 86400 \
    --evaluation-periods 1 \
    --alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name PublishInSuccessAlarm --state-reason
 "initializing" --state-value ALARM
```

4. Verify that the alarm appears in your [CloudWatch console](https://console.aws.amazon.com/cloudwatch/).

How can I be notified if my thing's shadow updates are being rejected each day?

1. Create an Amazon SNS topic named `things-shadow-updates-rejected`, and record its Amazon Resource Name (ARN). This procedure will refer to your topic's ARN as *sns-topic-arn* .

For more information on how to create an Amazon SNS notification, see Getting Started with Amazon SNS.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name UpdateThingShadowSuccessAlarm \
    --alarm-description "Alarm when my Things Shadow updates are getting rejected"
 \
    --namespace AWS/IoT \
    --metric-name UpdateThingShadow.Success \
    --dimensions Name=Protocol,Value=MQTT \
    --statistic Sum \
    --threshold 10 \
    --comparison-operator LessThanThreshold \
    --period 86400 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions sns-topic-arn
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --state-
reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name UpdateThingShadowSuccessAlarm --state-
reason "initializing" --state-value ALARM
```

4. Verify that the alarm appears in your CloudWatch console.

How can I create a CloudWatch alarm for jobs?

The Jobs service provides CloudWatch metrics for you to monitor your jobs. You can create CloudWatch alarms to monitor any Jobs metrics .

The following command creates a CloudWatch alarm to monitor the total number of failed job executions for Job *SampleOTAJob* and notifies you when more than 20 job executions have failed. The alarm monitors the Jobs metric `FailedJobExecutionTotalCount` by checking the reported value every 300 seconds. It is activated when a single reported value is greater than 20, meaning there were more than 20 failed job executions since the job started. When the alarm goes off, it sends a notification to the provided Amazon SNS topic.

```
aws cloudwatch put-metric-alarm \
    --alarm-name TotalFailedJobExecution-SampleOTAJob \
    --alarm-description "Alarm when total number of failed job execution exceeds the
 threshold for SampleOTAJob" \
    --namespace AWS/IoT \
    --metric-name FailedJobExecutionTotalCount \
    --dimensions Name=JobId,Value=SampleOTAJob \
    --statistic Sum \
    --threshold 20 \
    --comparison-operator GreaterThanThreshold \
    --period 300 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:<AWS_REGION>:<AWS_ACCOUNT_ID>:SampleOTAJob-has-too-
many-failed-job-ececutions
```

The following command creates a CloudWatch alarm to monitor the number of failed job executions for Job *`SampleOTAJob`* in a given period. It then notifies you when more than five job executions have failed during that period. The alarm monitors the Jobs metric `FailedJobExecutionCount` by checking the reported value every 3600 seconds. It is activated when a single reported value is greater than 5, meaning there were more than 5 failed job executions in the past hour. When the alarm goes off, it sends a notification to the provided Amazon SNS topic.

```
aws cloudwatch put-metric-alarm \
    --alarm-name FailedJobExecution-SampleOTAJob \
    --alarm-description "Alarm when number of failed job execution per hour exceeds the
 threshold for SampleOTAJob" \
    --namespace AWS/IoT \
    --metric-name FailedJobExecutionCount \
    --dimensions Name=JobId,Value=SampleOTAJob \
    --statistic Sum \
    --threshold 5 \
    --comparison-operator GreaterThanThreshold \
    --period 3600 \
    --unit Count \
    --evaluation-periods 1 \
```

```
    --alarm-actions arn:aws:sns:<AWS_REGION>:<AWS_ACCOUNT_ID>:SampleOTAJob-has-too-
many-failed-job-ececutions-per-hour
```

AWS IoT metrics and dimensions

When you interact with AWS IoT, the service sends metrics and dimensions to CloudWatch every minute. You can use AWS IoT, use the CloudWatch console or AWS CLI to view these metrics.

To view metrics using the CloudWatch console, open the CloudWatch console. In the navigation pane, choose **Metrics** and then choose **All metrics**. In the **Browse** tab, search for AWS IoT to view the list of metrics. Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

To view metrics using AWS CLI, run the following command.

```
aws cloudwatch list-metrics --namespace "AWS/IoT"
```

CloudWatch displays the following groups of metrics for AWS IoT:

- AWS IoT metrics
- AWS IoT Core credential provider metrics
- Authentication metrics
- Server certificate OCSP stapling metrics
- Rule metrics
- Rule action metrics
- HTTP action specific metrics
- Message broker metrics
- Device shadow metrics
- Logging metrics
- Jobs metrics
- Device Defender audit metrics
- Device Defender detect metrics
- Device provisioning metrics

- LoRaWAN metrics
- Fleet indexing metrics
- Dimensions for metrics

AWS IoT metrics

Metric	Description
AddThingToDynamicThingGroupsFailed	The number of failure events associated with adding a thing to a dynamic thing group. The DynamicThingGroupName dimension contains the name of the dynamic groups that failed to add things.
NumLogBatchesFailedToPublishThrottled	The singular batch of log events that has failed to publish due to throttling errors.
NumLogEventsFailedToPublishThrottled	The number of log events within the batch that have failed to publish due to throttling errors.

AWS IoT Core credential provider metrics

Metric	Description
CredentialExchangeSuccess	The number of successful AssumeRoleWithCertificate requests to AWS IoT Core credentials provider.

Authentication metrics

Note

The authentication metrics are displayed in the CloudWatch console under **Protocol Metrics**.

Metric	Description
`Connection.AuthNError`	The number of connection attempts which AWS IoT Core rejects due to authentication failures. This metric only considers connections that send a Server Name Indication (SNI) string matching an endpoint of your AWS account. This metric includes connection attempts from external sources such as internet scanning tools or probing activities. The `Protocol` dimension contains the protocol used to send the connection attempt.

Server certificate OCSP stapling metrics

Metric	Description
RetrieveOCSPStapleData.Success	The OCSP response has been received and processed successfully. This response will be included during the TLS handshake for the configured domain. The `DomainConfigurationName` dimension contains the name of configured domain with enabled server certificate OCSP stapling.

Rule metrics

Metric	Description
`ParseError`	The number of JSON parse errors that occurred in messages published on a topic on which a rule is listening. The `RuleName` dimension contains the name of the rule.
`RuleExecutionThrottled`	The number of messages throttled by the rules engine because of malicious behavior or because

Metric	Description
	the number of messages exceeds the rules engine's throttle limit. The `RuleName` dimension contains the name of the rule to be triggered.
`RuleNotFound`	The rule to be triggered could not be found. The `RuleName` dimension contains the name of the rule.
`RulesExecuted`	The number of AWS IoT rules executed.
`TopicMatch`	The number of incoming messages published on a topic on which a rule is listening. The `RuleName` dimension contains the name of the rule.

Rule action metrics

Metric	Description
`Failure`	The number of failed rule action invocations. The `RuleName` dimension contains the name of the rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.
`Success`	The number of successful rule action invocations. The `RuleName` dimension contains the name of the rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.
`ErrorActionFailure`	The number of failed error actions. The `RuleName` dimension contains the name of the rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.
`ErrorActionSuccess`	The number of successful error actions. The `RuleName` dimension contains the name of the

Metric	Description
	rule that specifies the action. The `ActionType` dimension contains the type of action that was invoked.

HTTP action specific metrics

Metric	Description
`HttpCode_Other`	Generated if the status code of the response from the downstream web service/application is not 2xx, 4xx or 5xx.
`HttpCode_4XX`	Generated if the status code of the response from the downstream web service/application is between 400 and 499.
`HttpCode_5XX`	Generated if the status code of the response from the downstream web service/application is between 500 and 599.
`HttpInvalidUrl`	Generated if an endpoint URL, after substitut ion templates are replaced, does not start with `https://`.
`HttpRequestTimeout`	Generated if the downstream web service/a pplication does not return response within request timeout limit. For more information, see Service Quotas.
`HttpUnknownHost`	Generated if the URL is valid, but the service does not exist or is unreachable.

Message broker metrics

Note

The message broker metrics are displayed in the CloudWatch console under **Protocol Metrics**.

Metric	Description
`Connect.AuthError`	The number of connection requests that could not be authorized by the message broker. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Connect.ClientError`	The number of connection requests rejected because the MQTT message did not meet the requirements defined in AWS IoT quotas . The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Connect.ClientIDThrottle`	The number of connection requests throttled because the client exceeded the allowed connect request rate for a specific client ID. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Connect.ServerError`	The number of connection requests that failed because an internal error occurred. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Connect.Success`	The number of successful connections to the message broker. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Connect.Throttle`	The number of connection requests that were throttled because the account exceeded the allowed

Metric	Description
	connect request rate. The `Protocol` dimension contains the protocol used to send the `CONNECT` message.
`Ping.Success`	The number of ping messages received by the message broker. The `Protocol` dimension contains the protocol used to send the ping message.
`PublishIn.AuthError`	The number of publish requests the message broker was unable to authorize. The `Protocol` dimension contains the protocol used to publish the message. HTTP Publish doesn't support this metric.
`PublishIn.ClientError`	The number of publish requests rejected by the message broker because the message did not meet the requirements defined in [AWS IoT quotas](). The `Protocol` dimension contains the protocol used to publish the message.
`PublishIn.ServerError`	The number of publish requests the message broker failed to process because an internal error occurred. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishIn.Success`	The number of publish requests successfully processed by the message broker. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishIn.Throttle`	The number of publish request that were throttled because the client exceeded the allowed inbound message rate. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message. HTTP Publish doesn't support this metric.

Metric	Description
`PublishOut.AuthError`	The number of publish requests made by the message broker that could not be authorized by AWS IoT. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishOut.ClientError`	The number of publish requests made by the message broker that were rejected because the message did not meet the requirements defined in [AWS IoT quotas](). The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishOut.Success`	The number of publish requests successfully made by the message broker. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishOut.Throttle`	The number of publish requests that were throttled because the client exceeded the allowed outbound message rate. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishRetained.AuthError`	The number of publish requests with the `RETAIN` flag set that the message broker was unable to authorize. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishRetained.ServerError`	The number of retained publish requests the message broker failed to process because an internal error occurred. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.
`PublishRetained.Success`	The number of publish requests with the `RETAIN` flag set that were successfully processed by the message broker. The `Protocol` dimension contains the protocol used to send the `PUBLISH` message.

Metric	Description
PublishRetained.Throttle	The number of publish requests with the RETAIN flag set that were throttled because the client exceeded the allowed inbound message rate. The Protocol dimension contains the protocol used to send the PUBLISH message.
Queued.Success	The number of stored messages that were successfully processed by the message broker for clients that were disconnected from their persistent session. Messages with a QoS of 1 are stored while a client with a persistent session is disconnected.
Queued.Throttle	The number of messages that couldn't be stored and were throttled while clients with persistent sessions were disconnected. This occurs when clients exceed the Queued messages per second per account limit. Messages with a QoS of 1 are stored while a client with a persistent session is disconnected.
Queued.ServerError	The number of messages that haven't been stored for a persistent session because of an internal error. When clients with a persistent session are disconnected, messages with a Quality of Service (QoS) of 1 are stored.
Subscribe.AuthError	The number of subscription requests made by a client that could not be authorized. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.
Subscribe.ClientError	The number of subscribe requests that were rejected because the SUBSCRIBE message did not meet the requirements defined in AWS IoT quotas. The Protocol dimension contains the protocol used to send the SUBSCRIBE message.

Metric	Description
`Subscribe.ServerError`	The number of subscribe requests that were rejected because an internal error occurred. The `Protocol` dimension contains the protocol used to send the `SUBSCRIBE` message.
`Subscribe.Success`	The number of subscribe requests that were successfully processed by the message broker. The `Protocol` dimension contains the protocol used to send the `SUBSCRIBE` message.
`Subscribe.Throttle`	The number of subscribe requests that were throttled because the allowed subscribe request rate limits were exceeded for your AWS account. These limits include Subscriptions per second per account, Subscriptions per account, and Subscript ions per connection described in AWS IoT Core message broker and protocol limits and quotas. The `Protocol` dimension contains the protocol used to send the `SUBSCRIBE` message.
`Throttle.Exceeded`	This metric will display in CloudWatch when an MQTT client is throttled on packets per second per connection level limits. This metric doesn't apply to HTTP connections.
`Unsubscribe.ClientError`	The number of unsubscribe requests that were rejected because the `UNSUBSCRIBE` message did not meet the requirements defined in AWS IoT quotas . The `Protocol` dimension contains the protocol used to send the `UNSUBSCRIBE` message.
`Unsubscribe.ServerError`	The number of unsubscribe requests that were rejected because an internal error occurred. The `Protocol` dimension contains the protocol used to send the `UNSUBSCRIBE` message.

Metric	Description
`Unsubscribe.Success`	The number of unsubscribe requests that were successfully processed by the message broker. The `Protocol` dimension contains the protocol used to send the `UNSUBSCRIBE` message.
`Unsubscribe.Throttle`	The number of unsubscribe requests that were rejected because the client exceeded the allowed unsubscribe request rate. The `Protocol` dimension contains the protocol used to send the `UNSUBSCRIBE` message.

Device shadow metrics

> **Note**
>
> The device shadow metrics are displayed in the CloudWatch console under **Protocol Metrics**.

Metric	Description
`DeleteThingShadow.Accepted`	The number of `DeleteThingShadow` requests processed successfully. The `Protocol` dimension contains the protocol used to make the request.
`GetThingShadow.Accepted`	The number of `GetThingShadow` requests processed successfully. The `Protocol` dimension contains the protocol used to make the request.
`ListThingShadow.Accepted`	The number of `ListThingShadow` requests processed successfully. The `Protocol` dimension contains the protocol used to make the request.

Metric	Description
`UpdateThingShadow.Accepted`	The number of `UpdateThingShadow` requests processed successfully. The `Protocol` dimension contains the protocol used to make the request.

Logging metrics

Metric	Description
`CloudwatchLogs:LogGroupCreationFailed`	The number of `CreateLogGroup` request failures. AWS IoT Logging creates CloudWatch log group if it doesn't exist. The `LogGroup` dimension contains the log group used to make the request.
`CloudwatchLogs:LogStreamCreationFailed`	The number of `CreateLogStream` request failures. AWS IoT Logging creates CloudWatch log stream to push log events. The `LogGroup` dimension contains the log group used to make the request.
`CloudwatchLogs:PutRetentionPolicyFailed`	The number of `PutRetentionPolicy` request failures. AWS IoT Logging creates CloudWatch log group if it doesn't exist, and the retention policy is set as 30 days. The `LogGroup` dimension contains the log group used to make the request.
`CloudwatchLogs:PutLogEventsFailed`	The number of `PutLogEvents` request failures. The `LogGroup` dimension contains the CloudWatch log group used to make the request.

Jobs metrics

Metric	Description
CanceledJobExecutionCount	The number of job executions whose status has changed to CANCELED within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics.) The JobId dimension contains the ID of the job.
CanceledJobExecutionTotalCount	The total number of job executions whose status is CANCELED for the given job. The JobId dimension contains the ID of the job.
ClientErrorCount	The number of client errors generated while executing the job. The JobId dimension contains the ID of the job.
FailedJobExecutionCount	The number of job executions whose status has changed to FAILED within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics.) The JobId dimension contains the ID of the job.
FailedJobExecutionTotalCount	The total number of job executions whose status is FAILED for the given job. The JobId dimension contains the ID of the job.
InProgressJobExecutionCount	The number of job executions whose status has changed to IN_PROGRESS within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics.) The JobId dimension contains the ID of the job.

Metric	Description
`InProgressJobExecutionTotalCount`	The total number of job executions whose status is `IN_PROGRESS` for the given job. The `JobId` dimension contains the ID of the job.
`RejectedJobExecutionTotalCount`	The total number of job executions whose status is `REJECTED` for the given job. The `JobId` dimension contains the ID of the job.
`RemovedJobExecutionTotalCount`	The total number of job executions whose status is `REMOVED` for the given job. The `JobId` dimension contains the ID of the job.
`QueuedJobExecutionCount`	The number of job executions whose status has changed to `QUEUED` within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see [Amazon CloudWatch Metrics](#).) The `JobId` dimension contains the ID of the job.
`QueuedJobExecutionTotalCount`	The total number of job executions whose status is `QUEUED` for the given job. The `JobId` dimension contains the ID of the job.
`RejectedJobExecutionCount`	The number of job executions whose status has changed to `REJECTED` within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see [Amazon CloudWatch Metrics](#).) The `JobId` dimension contains the ID of the job.
`RemovedJobExecutionCount`	The number of job executions whose status has changed to `REMOVED` within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see [Amazon CloudWatch Metrics](#).) The `JobId` dimension contains the ID of the job.

Metric	Description
`ServerErrorCount`	The number of server errors generated while executing the job. The `JobId` dimension contains the ID of the job.
`SuccededJobExecutionCount`	The number of job executions whose status has changed to `SUCCESS` within a time period that is determined by CloudWatch. (For more information about CloudWatch metrics, see Amazon CloudWatch Metrics.) The `JobId` dimension contains the ID of the job.
`SuccededJobExecutionTotalCount`	The total number of job executions whose status is `SUCCESS` for the given job. The `JobId` dimension contains the ID of the job.

Device Defender audit metrics

Metric	Description
`NonCompliantResources`	The number of resources that were found to be noncompliant with a check. The system reports the number of resources that were out of compliance for each check of each audit performed.
`ResourcesEvaluated`	The number of resources that were evaluated for compliance. The system reports the number of resources that were evaluated for each check of each audit performed.
`MisconfiguredDeviceDefenderNotification`	Notifies you when your SNS configuration for AWS IoT Device Defender is misconfigured. Dimensions

Device Defender detect metrics

Metric	Description
`NumOfMetricsExported`	The number of metrics exported for a cloud-side, device-side, or custom metric. The system reports the number of metrics exported for the account, for a specific metric. This metric is available only for customers using metrics export.
`NumOfMetricsSkipped`	The number of metrics skipped for a cloud-side, device-side, or custom metric. The system reports the number of metrics skipped for the account, for a specific metric due to insufficient permissions provided to Device Defender Detect to publish to the mqtt topic. This metric is available only for customers using metrics export.
`NumOfMetricsExceedingSizeLimit`	The number of metrics skipped for export for a cloud-side, device-side, or custom metric due to size exceeding MQTT message size constraints. The system reports the number of metrics skipped for export for the account, for a specific metric due to size exceeding MQTT message size constraints. This metric is available only for customers using metrics export.
`Violations`	The number of new violations of security profile behaviors that have been found since the last time an evaluation was performed. The system reports the number of new violations for the account, for a specific security profile, and for a specific behavior of a specific security profile.
`ViolationsCleared`	The number of violations of security profile behaviors that have been resolved since the last time an evaluation was performed. The system reports the number of resolved violations for the

Metric	Description
	account, for a specific security profile, and for a specific behavior of a specific security profile.
`ViolationsInvalidated`	The number of violations of security profile behaviors for which information is no longer available since the last time an evaluation was performed (because the reporting device stopped reporting, or is no longer being monitored for some reason). The system reports the number of invalidated violations for the entire account, for a specific security profile, and for a specific behavior of a specific security profile.
`MisconfiguredDeviceDefenderNotification`	Notifies you when your SNS configuration for AWS IoT Device Defender is misconfigured. Dimensions

Device provisioning metrics

AWS IoT Fleet provisioning metrics

Metric	Description
`ApproximateNumberOfThingsRegistered`	The count of things that have been registered by Fleet Provisioning. While the count is generally accurate, the distributed architecture of AWS IoT Core makes it difficult to maintain a precise count of registered things. The statistic to use for this metric is: • **Max** to report the total number of things that have been registered. For a count of things registered during the CloudWatch aggregation

Metric	Description
	window, see the `RegisterThingFailed` metric. Dimensions: ClaimCertificateId
`CreateKeysAndCertificateFailed`	The number of failures that occurred by calls to the `CreateKeysAndCertificate` MQTT API. The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of certificates created and registered during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour. The statistics available for this metric are: • **Sum** to report the number of failed calls. • **SampleCount** to report the total number of successful and failed calls.
`CreateCertificateFromCsrFailed`	The number of failures that occurred by calls to the `CreateCertificateFromCsr` MQTT API. The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of things registere d during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour. The statistics available for this metric are: • **Sum** to report the number of failed calls. • **SampleCount** to report the total number of successful and failed calls.

Metric	Description
`RegisterThingFailed`	The number of failures that occurred by calls to the `RegisterThing` MQTT API. The metric is emitted in both Success (value = 0) and Failure (value = 1) cases. This metric can be used to track the number of things registered during the CloudWatch-supported aggregation windows, such as 5 min. or 1 hour. For the total number of things registered , see the `ApproximateNumberOfThingsRegistered` metric. The statistics available for this metric are: • **Sum** to report the number of failed calls. • **SampleCount** to report the total number of successful and failed calls. Dimensions: TemplateName

Just-in-time provisioning metrics

Metric	Description
`ProvisionThing.ClientError`	The number of times a device failed to provision due to a client error. For example, the policy specified in the template did not exist.
`ProvisionThing.ServerError`	The number of times a device failed to provision due to a server error. Customers can retry to provision the device after waiting and they can contact AWS IoT if the issue remains the same.
`ProvisionThing.Success`	The number of times a device was successfully provisioned.

LoRaWAN metrics

The following table shows the metrics for AWS IoT Core for LoRaWAN. For more information, see AWS IoT Core for LoRaWAN metrics.

AWS IoT Core for LoRaWAN metrics

Metric	Description
Active devices/gateways	The number of active LoRaWAN devices and gateways in your account.
Uplink message count	The number of uplink messages that are sent within a specified time duration for all active gateways and devices in your AWS account. Uplink messages are messages that are sent from your device to AWS IoT Core for LoRaWAN.
Downlink message count	The number of downlink messages that are sent within a specified time duration for all active gateways and devices in your AWS account. Downlink messages are messages that are sent from AWS IoT Core for LoRaWAN to your device.
Message loss rate	After you've added your device and connected to AWS IoT Core for LoRaWAN, your device can initiate an uplink message to start exchanging messages with the cloud. You can use this metric to then track the rate of uplink messages that are lost.
Join metrics	After you've added your device and gateway, you perform a join procedure so that your device can send uplink data and communicate with AWS IoT Core for LoRaWAN. You can use this metric to obtain information about join metrics for all active devices in your AWS account.
Average received signal strength indicator (RSSI)	You can use this metric to monitor the average RSSI (Received signal strength indicator) within the

Metric	Description
	specified time duration. RSSI is a measurement that indicates if the signal is strong enough for a good wireless connection. This value is negative and must be closer to zero for a strong connection.
Average signal to noise ratio (SNR)	You can use this metric to monitor the average SNR (Signal-to-noise ratio) within the specified time duration. SNR is a measurement that indicates if the received signal is strong enough compared to the noise level for a good wireless connection. The SNR value is positive and must be greater than zero to indicate that the signal power is stronger than the noise power.
Gateway availability	You can use this metric to obtain information about the availability of this gateway within a specified time duration. This metric displays the websocket connection time of this gateway for a specified time duration.

Just-in-time provisioning metrics

Metric	Description
`ProvisionThing.ClientError`	The number of times a device failed to provision due to a client error. For example, the policy specified in the template did not exist.
`ProvisionThing.ServerError`	The number of times a device failed to provision due to a server error. Customers can retry to provision the device after waiting and they can contact AWS IoT if the issue remains the same.
`ProvisionThing.Success`	The number of times a device was successfully provisioned.

Fleet indexing metrics

AWS IoT fleet indexing metrics

Metric	Description
NamedShadowCountForDynamicGroupQueryLimitExceeded	A maximum of 25 named shadows per thing are processed for query terms that are not data source specific in dynamic thing groups. When this limit is breached for a thing, the NamedShadowCountForDynamicGroupQueryLimitExceeded event type will be emitted.

Dimensions for metrics

Metrics use the namespace and provide metrics for the following dimensions

Dimension	Description
ActionType	The action type specified by the rule that triggered the request.
BehaviorName	The name of the Device Defender Detect security profile behavior that is being monitored.
ClaimCertificateId	The certificateId of the claim used to provision the devices.
CheckName	The name of the Device Defender audit check whose results are being monitored.
JobId	The ID of the job whose progress or message connection success/failure is being monitored.
Protocol	The protocol used to make the request. Valid values are: MQTT or HTTP
RuleName	The name of the rule triggered by the request.

Dimension	Description
ScheduledAuditName	The name of the Device Defender scheduled audit whose check results are being monitored. This has the value OnDemand if the results reported are for an audit that was performed on demand.
SecurityProfileName	The name of the Device Defender Detect security profile whose behaviors are being monitored.
TemplateName	The name of the provisioning template.
SourceArn	Refers to the security profile for detect or the account arn for audit.
RoleArn	Refers to the role Device Defender attempted to assume.
TopicArn	Refers to the SNS topic Device Defender attempted to publish to.

Dimension	Description
`Error`	Gives a short description of the Error received while attempting to publish to the SNS topic. Possible values are: • "KMSKeyNotFound": indicates the KMS key does not exist for the topic. • "InvalidTopicName": indicates the SNS Topic is not valid. • "KMSAccessDenied": indicates that the role does not have permissions to the KMS key for the Topic. • "AuthorizationError": indicates that the role provided does not authorize Device Defender to publish to the SNS topic. • "SNSTopicNotFound": indicates the provided SNS topic does not exist. • "FailureToAssumeRole": indicates that the role provided does not authorize Device Defender to assume the role. • "CrossRegionSNSTopic": indicates that the SNS topic exists in a different region.

Monitor AWS IoT using CloudWatch Logs

When AWS IoT logging is enabled, AWS IoT sends progress events about each message as it passes from your devices through the message broker and rules engine. In the CloudWatch console, CloudWatch logs appear in a log group named **AWSIotLogs**.

For more information about CloudWatch Logs, see CloudWatch Logs. For information about supported AWS IoT CloudWatch Logs, see CloudWatch Logs AWS IoT log entries .

Viewing AWS IoT logs in the CloudWatch console

Note

The `AWSIotLogsV2` log group is not visible in the CloudWatch console until:

- You've enabled logging in AWS IoT. For more info on how to enable logging in AWS IoT, see Configure AWS IoT logging
- Some log entries have been written by AWS IoT operations.

To view your AWS IoT logs in the CloudWatch console

1. Browse to https://console.aws.amazon.com/cloudwatch/. In the navigation pane, choose **Log groups**.
2. In the **Filter** text box, enter **`AWSIotLogsV2`** , and then press Enter.
3. Double-click the `AWSIotLogsV2` log group.
4. Choose **Search All**. A complete list of the AWS IoT logs generated for your account is displayed.
5. Choose the expand icon to look at an individual stream.

You can also enter a query in the **Filter events** text box. Here are some interesting queries to try:

- `{ $.logLevel = "INFO" }`

  Find all logs that have a log level of `INFO`.
- `{ $.status = "Success" }`

  Find all logs that have a status of `Success`.
- `{ $.status = "Success" && $.eventType = "GetThingShadow" }`

  Find all logs that have a status of `Success` and an event type of `GetThingShadow`.

For more information about creating filter expressions, see CloudWatch Logs Queries.

CloudWatch Logs AWS IoT log entries

Each component of AWS IoT generates its own log entries. Each log entry has an `eventType` that specifies the operation that caused the log entry to be generated. This section describes the log entries generated by the following AWS IoT components.

Topics

Message broker log entries

The AWS IoT message broker generates log entries for the following events:

Topics

Connect log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Connect` when an MQTT client connects.

Connect log entry example

```
{
    "timestamp": "2017-08-10 15:37:23.476",
    "logLevel": "INFO",
    "traceId": "20b23f3f-d7f1-feae-169f-82263394fbdb",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Connect",
    "protocol": "MQTT",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the [Common CloudWatch Logs attributes](#) ,`Connect` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

Disconnect log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Disconnect` when an MQTT client disconnects.

Disconnect log entry example

```
{
    "timestamp": "2017-08-10 15:37:23.476",
    "logLevel": "INFO",
    "traceId": "20b23f3f-d7f1-feae-169f-82263394fbdb",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Disconnect",
    "protocol": "MQTT",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490,
    "reason": "DUPLICATE_CLIENT_ID",
    "details": "A new connection was established with the same client ID",
    "disconnectReason": "CLIENT_INITIATED_DISCONNECT"
}
```

In addition to the Common CloudWatch Logs attributes ,`Disconnect` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

reason

The reason why the client is disconnecting.

details

A brief explanation of the error.

disconnectReason

The reason why the client is disconnecting.

DeleteConnection log entry

The AWS IoT message broker generates a log entry with an `eventType` of `DeleteConnection` when an MQTT client connection is deleted.

DeleteConnection log entry example

```
{
    "timestamp": "2025-08-09 15:37:23.476",
    "logLevel": "INFO",
    "traceId": "20b23f3f-d7f1-feae-169f-82263394fbdb",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "DeleteConnection",
    "protocol": "HTTP",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the Common CloudWatch Logs attributes ,`DeleteConnection` log entries contain the following attributes:

clientId

The ID of the client that will be disconnected.

principalId

The ID of the principal making the request. For information about how to identify the principal using the principal ID, see Compare IAM identities and credentials in the *IAM User Guide*.

protocol

The protocol used to make the request. The valid value is HTTP.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

GetRetainedMessage log entry

The AWS IoT message broker generates a log entry with an `eventType` of `GetRetainedMessage` when `GetRetainedMessage` is called.

GetRetainedMessage log entry example

```
{
    "timestamp": "2017-08-07 18:47:56.664",
    "logLevel": "INFO",
    "traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "GetRetainedMessage",
    "protocol": "HTTP",
    "topicName": "a/b/c",
    "qos": "1",
    "lastModifiedDate": "2017-08-07 18:47:56.664"
}
```

In addition to the Common CloudWatch Logs attributes ,`GetRetainedMessage` log entries contain the following attributes:

lastModifiedDate

The Epoch date and time, in milliseconds, when the retained message was stored by AWS IoT.

protocol

The protocol used to make the request. Valid value: HTTP.

qos

The Quality of Service (QoS) level used in the publish request. Valid values are `0` or 1.

topicName

The name of the subscribed topic.

ListRetainedMessage log entry

The AWS IoT message broker generates a log entry with an `eventType` of `ListRetainedMessage` when [`ListRetainedMessages`]() is called.

ListRetainedMessage log entry example

```
{
    "timestamp": "2017-08-07 18:47:56.664",
    "logLevel": "INFO",
    "traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "ListRetainedMessage",
    "protocol": "HTTP"
}
```

In addition to the [Common CloudWatch Logs attributes]() ,`ListRetainedMessage` log entries contains the following attribute:

protocol

The protocol used to make the request. Valid value: `HTTP`.

Publish-In log entry

When the AWS IoT message broker receives an MQTT message, it generates a log entry with an `eventType` of `Publish-In`.

Publish-In log entry example

```
{
        "timestamp": "2017-08-10 15:39:30.961",
        "logLevel": "INFO",
```

```
        "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
        "accountId": "123456789012",
        "status": "Success",
        "eventType": "Publish-In",
        "protocol": "MQTT",
        "topicName": "$aws/things/MyThing/shadow/get",
        "clientId": "abf27092886e49a8a5c1922749736453",
        "principalId":
"145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
        "sourceIp": "205.251.233.181",
        "sourcePort": 13490,
        "retain": "True"
    }
```

In addition to the Common CloudWatch Logs attributes ,`Publish-In` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

retain

The attribute used when a message has the RETAIN flag set with a value of `True`. If the message doesn't have the RETAIN flag set, this attribute doesn't appear in the log entry. For more information, see MQTT retained messages .

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

topicName

The name of the subscribed topic.

Publish-Out log entry

When the message broker publishes an MQTT message, it generates a log entry with an `eventType` of `Publish-Out`

Publish-Out log entry example

```
{
    "timestamp": "2017-08-10 15:39:30.961",
    "logLevel": "INFO",
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Publish-Out",
    "protocol": "MQTT",
    "topicName": "$aws/things/MyThing/shadow/get",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the Common CloudWatch Logs attributes ,`Publish-Out` log entries contain the following attributes:

clientId

The ID of the subscribed client that receives messages on that MQTT topic.

principalId

The ID of the principal making the request.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

topicName

The name of the subscribed topic.

Queued log entry

When a device with a persistent session is disconnected, the MQTT message broker stores the device's messages and AWS IoT generates log entries with an eventType of `Queued`. For more information about MQTT persistent sessions, see MQTT persistent sessions .

Queued server error log entry example

```
{
    "timestamp": "2022-08-10 15:39:30.961",
    "logLevel": "ERROR",
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
    "accountId": "123456789012",
    "topicName": "$aws/things/MyThing/get",
    "clientId": "123123123",
    "qos": "1",
    "protocol": "MQTT",
    "eventType": "Queued",
    "status": "Failure",
    "details": "Server Error"
}
```

In addition to the Common CloudWatch Logs attributes ,`Queued` server error log entries contain the following attributes:

clientId

The ID of the client to which the message is queued.

details

`Server Error`

A server error prevented the message from being stored.

protocol

The protocol used to make the request. The value will always be `MQTT`.

qos

The Quality of Service (QoS) level of the request. The value will always be 1 because the messages with QoS of 0 aren't stored.

topicName

The name of the subscribed topic.

Queued success log entry example

```
{
    "timestamp": "2022-08-10 15:39:30.961",
    "logLevel": "INFO",
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
    "accountId": "123456789012",
    "topicName": "$aws/things/MyThing/get",
    "clientId": "123123123",
    "qos": "1",
    "protocol": "MQTT",
    "eventType": "Queued",
    "status": "Success"
}
```

In addition to the Common CloudWatch Logs attributes ,`Queued` success log entries contain the following attributes:

clientId

The ID of the client to which the message is queued.

protocol

The protocol used to make the request. The value will always be `MQTT`.

qos

The Quality of Service (QoS) level of the request. The value will always be 1 because the messages with QoS of 0 aren't stored.

topicName

The name of the subscribed topic.

Queued throttled log entry example

```
{
    "timestamp": "2022-08-10 15:39:30.961",
    "logLevel": "ERROR",
    "traceId": "672ec480-31ce-fd8b-b5fb-22e3ac420699",
    "accountId": "123456789012",
    "topicName": "$aws/things/MyThing/get",
    "clientId": "123123123",
    "qos": "1",
    "protocol": "MQTT",
    "eventType": "Queued",
    "status": "Failure",
    "details": "Throttled while queueing offline message"
}
```

In addition to the Common CloudWatch Logs attributes ,`Queued` throttled log entries contain the following attributes:

clientId

The ID of the client to which the message is queued.

details

`Throttled while queueing offline message`

The client exceeded the `Queued messages per second per account` limit, so the message wasn't stored.

protocol

The protocol used to make the request. The value will always be `MQTT`.

qos

The Quality of Service (QoS) level of the request. The value will always be 1 because the messages with QoS of 0 aren't stored.

topicName

The name of the subscribed topic.

Subscribe log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Subscribe` when an MQTT client subscribes to a topic.

MQTT 3 Subscribe log entry example

```
{
    "timestamp": "2017-08-10 15:39:04.413",
    "logLevel": "INFO",
    "traceId": "7aa5c38d-1b49-3753-15dc-513ce4ab9fa6",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Subscribe",
    "protocol": "MQTT",
    "topicName": "$aws/things/MyThing/shadow/#",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the Common CloudWatch Logs attributes ,`Subscribe` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

protocol

The protocol used to make the request. The value will always be `MQTT`.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

topicName

The name of the subscribed topic.

MQTT 5 Subscribe log entry example

```
{
 "timestamp": "2022-11-30 16:24:15.628",
 "logLevel": "INFO",
 "traceId": "7aa5c38d-1b49-3753-15dc-513ce4ab9fa6",
 "accountId": "123456789012",
 "status": "Success",
 "eventType": "Subscribe",
 "protocol": "MQTT",
 "topicName": "test/topic1,$invalid/reserved/topic",
 "subscriptions": [
  {
   "topicName": "test/topic1",
   "reasonCode": 1
  },
  {
   "topicName": "$invalid/reserved/topic",
   "reasonCode": 143
  }
 ],
 "clientId": "abf27092886e49a8a5c1922749736453",
 "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
 "sourceIp": "205.251.233.181",
 "sourcePort": 13490
}
```

For MQTT 5 Subscribe operations, in addition to the Common CloudWatch Logs attributes and the MQTT 3 Subscribe log entry attributes, MQTT 5 `Subscribe` log entries contain the following attribute:

subscriptions

A list of mappings between the requested topics in the Subscribe request and the individual MQTT 5 reason code. For more information, see MQTT reason codes.

Unsubscribe log entry

The AWS IoT message broker generates a log entry with an `eventType` of `Unsubscribe` when an MQTT client unsubscribes to an MQTT topic.

MQTT unsubscribe log entry example

```
{
    "timestamp": "2024-08-20 22:53:32.844",
    "logLevel": "INFO",
    "traceId": "db6bd09a-2c3f-1cd2-27cc-fd6b1ce03b58",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "Unsubscribe",
    "protocol": "MQTT",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "sourceIp": "205.251.233.181",
    "sourcePort": 13490
}
```

In addition to the Common CloudWatch Logs attributes ,`Unsubscribe` log entries contain the following attributes:

protocol

The protocol used to make the request. The value will always be `MQTT`.

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

sourceIp

The IP address where the request originated.

sourcePort

The port where the request originated.

Server certificate OCSP log entries

AWS IoT Core generates log entries for the following event:

Topics

- RetrieveOCSPStapleData log entry
- RetrieveOCSPStapleData log entry for private endpoints

RetrieveOCSPStapleData log entry

AWS IoT Core generates a log entry with an `eventType` of `RetrieveOCSPStapleData` when the server retrieves the OCSP staple data.

RetrieveOCSPStapleData log entry examples

The following is a log entry example of `Success`.

```
{
 "timestamp": "2024-01-30 15:39:30.961",
 "logLevel": "INFO",
 "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
 "accountId": "123456789012",
 "status": "Success",
 "eventType": "RetrieveOCSPStapleData",
 "domainConfigName": "test-domain-config-name",
 "connectionDetails": {
  "httpStatusCode": "200",
  "ocspResponderUri": "http://ocsp.example.com",
  "sourceIp": "205.251.233.181",
  "targetIp": "250.15.5.3"
 },
 "ocspRequestDetails": {
  "requesterName": "iot.amazonaws.com",
  "requestCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
 },
 "ocspResponseDetails": {
  "responseCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
  "ocspResponseStatus": "successful",
  "certStatus": "good",
```

```
  "signature":
 "4C:6F:63:61:6C:20:52:65:73:70:6F:6E:64:65:72:20:53:69:67:6E:61:74:75:72:65",
  "thisUpdateTime": "Jan 31 01:21:02 2024 UTC",
  "nextUpdateTime": "Feb 02 00:21:02 2024 UTC",
  "producedAtTime": "Jan 31 01:37:03 2024 UTC",
  "stapledDataPayloadSize": "XXX"
 }
}
```

The following is a log entry example of `Failure`.

```
{
 "timestamp": "2024-01-30 15:39:30.961",
 "logLevel": "ERROR",
 "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
 "accountId": "123456789012",
 "status": "Failure",
 "reason": "A non 2xx HTTP response was received from the OCSP responder.",
 "eventType": "RetrieveOCSPStapleData",
 "domainConfigName": "test-domain-config-name",
 "connectionDetails": {
  "httpStatusCode": "444",
  "ocspResponderUri": "http://ocsp.example.com",
  "sourceIp": "205.251.233.181",
  "targetIp": "250.15.5.3"
 },
 "ocspRequestDetails": {
  "requesterName": "iot.amazonaws.com",
  "requestCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
 }
}
```

For the `RetrieveOCSPStaple` operation, in addition to the [Common CloudWatch Logs attributes](#), the log entries contain the following attributes:

reason

The reason why the operation fails.

domainConfigName

The name of your domain configuration.

connectionDetails

A brief explanation of the connection details.

- httpStatusCode

  HTTP status codes that are returned by the OCSP responder in response to the client's request made to the server.
- ocspResponderUri

  The OCSP responder URI that AWS IoT Core fetches from the server certificate.
- sourceIp

  The source IP address of the AWS IoT Core server.
- targetIp

  The target IP address of the OCSP responder.

ocspRequestDetails

Details of the OCSP request.

- requesterName

  The identifier for the AWS IoT Core server that sends a request to the OCSP responder.
- requestCertId

  The certificate ID of the request. This is the ID of the certificate for which the OCSP response is being requested.

ocspResponseDetails

Details of the OCSP response.

- responseCertId

  The certificate ID of the OCSP response.
- ocspResponseStatus

  The status of the OCSP response.
- certStatus

  The status of the certificate.

- signature

  The signature that's applied to the response by a trusted entity.
- thisUpdateTime

  The time at which the status being indicated is known to be correct.
- nextUpdateTime

  The time at or before which newer information will be available about the status of the certificate.
- producedAtTime

  The time at which the OCSP responder signed this response.
- stapledDataPayloadSize

  The payload size of the stapled data.

RetrieveOCSPStapleData log entry for private endpoints

AWS IoT Core generates a log entry with an `eventType` of `RetrieveOCSPStapleData` when the server retrieves the OCSP staple data.

RetrieveOCSPStapleData log entry examples for private endpoints

The following is a log entry example of `Success`.

```
{
 "timestamp": "2024-01-30 15:39:30.961",
 "logLevel": "INFO",
 "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
 "accountId": "123456789012",
 "status": "Success",
 "eventType": "RetrieveOCSPStapleData",
 "domainConfigName": "test-domain-config-name",
        "lambdaDetails": {
        "lambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
        "sourceArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
testDomainConfigure/6bzfg"
      },
        "authorizedResponderArn": "arn:aws:acm:us-west-2:123456789012:certificate/
certificate_ID",
```

```
 "ocspRequestDetails": {
  "requesterName": "iot.amazonaws.com",
  "requestCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
 },
 "ocspResponseDetails": {
  "responderId": "04:C1:3F:8F:27:D6:49:13:F8:DE:B2:36:9D:85:8E:F8:31:3B:A6:D0"
              "responseCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
  "ocspResponseStatus": "successful",
  "certStatus": "good",
  "signature":
 "4C:6F:63:61:6C:20:52:65:73:70:6F:6E:64:65:72:20:53:69:67:6E:61:74:75:72:65",
  "thisUpdateTime": "Jan 31 01:21:02 2024 UTC",
  "nextUpdateTime": "Feb 02 00:21:02 2024 UTC",
  "producedAtTime": "Jan 31 01:37:03 2024 UTC",
  "stapledDataPayloadSize": "XXX"
 }
}
```

The following is a log entry example of `Failure`.

```
{
 "timestamp": "2024-01-30 15:39:30.961",
 "logLevel": "ERROR",
 "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
 "accountId": "123456789012",
 "status": "Failure",
 "reason": "The payload returned by the Lambda function exceeds the maximum response
 size of 7 kilobytes.",
 "eventType": "RetrieveOCSPStapleData",
 "domainConfigName": "test-domain-config-name",
        "lambdaDetails": {
        "lambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
        "sourceArn": "arn:aws:iot:us-west-2:123456789012:domainconfiguration/
testDomainConfigure/6bzfg"
      },
        "authorizedResponderArn": "arn:aws:acm:us-west-2:123456789012:certificate/
certificate_ID",
  "ocspRequestDetails": {
  "requesterName": "iot.amazonaws.com",
  "requestCertId":
 "30:3A:30:09:06:05:2B:0E:03:02:1A:05:00:04:14:9C:FF:90:A1:97:B0:4D:6C:01:B9:69:96:D8:3E:E7:A2:
```

```
  }
}
```

For the `RetrieveOCSPStaple` operation, in addition to the Common CloudWatch Logs attributes and the attributes in RetrieveOCSPStapleData log entry, the log entries for private endpoints contain the following attributes:

lambdaDetails

Details of the Lambda function.

- lambdaArn

  The ARN of the Lambda function.
- sourceArn

  The ARN of the domain configuration.

authorizedResponderArn

The ARN of the authorizer responder if there is one configured in the domain configuration.

Device Shadow log entries

The AWS IoT Device Shadow service generates log entries for the following events:

Topics

DeleteThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of `DeleteThingShadow` when a request to delete a device's shadow is received.

DeleteThingShadow log entry example

```
{
```

```
    "timestamp": "2017-08-07 18:47:56.664",
    "logLevel": "INFO",
    "traceId": "1a60d02e-15b9-605b-7096-a9f584a6ad3f",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "DeleteThingShadow",
    "protocol": "MQTT",
    "deviceShadowName": "Jack",
    "topicName": "$aws/things/Jack/shadow/delete"
}
```


In addition to the Common CloudWatch Logs attributes ,`DeleteThingShadow` log entries contain the following attributes:

deviceShadowName

The name of the shadow to update.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The name of the topic on which the request was published.

GetThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of `GetThingShadow` when a get request for a shadow is received.

GetThingShadow log entry example


```
{
    "timestamp": "2017-08-09 17:56:30.941",
    "logLevel": "INFO",
    "traceId": "b575f19a-97a2-cf72-0ed0-c64a783a2504",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "GetThingShadow",
    "protocol": "MQTT",
    "deviceShadowName": "MyThing",
    "topicName": "$aws/things/MyThing/shadow/get"
```

```
}
```

In addition to the Common CloudWatch Logs attributes ,`GetThingShadow` log entries contain the following attributes:

deviceShadowName

The name of the requested shadow.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The name of the topic on which the request was published.

UpdateThingShadow log entry

The Device Shadow service generates a log entry with an `eventType` of `UpdateThingShadow` when a request to update a device's shadow is received.

UpdateThingShadow log entry example

```
{
    "timestamp": "2017-08-07 18:43:59.436",
    "logLevel": "INFO",
    "traceId": "d0074ba8-0c4b-a400-69df-76326d414c28",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "UpdateThingShadow",
    "protocol": "MQTT",
    "deviceShadowName": "Jack",
    "topicName": "$aws/things/Jack/shadow/update"
}
```

In addition to the Common CloudWatch Logs attributes ,`UpdateThingShadow` log entries contain the following attributes:

deviceShadowName

The name of the shadow to update.

protocol

The protocol used to make the request. Valid values are MQTT or HTTP.

topicName

The name of the topic on which the request was published.

Rules engine log entries

The AWS IoT rules engine generates logs for the following events:

Topics

FunctionExecution log entry

The rules engine generates a log entry with an `eventType` of `FunctionExecution` when a rule's SQL query calls an external function. An external function is called when a rule's action makes an HTTP request to AWS IoT or another web service (for example, calling `get_thing_shadow` or `machinelearning_predict`).

FunctionExecution log entry example

```
{
    "timestamp": "2017-07-13 18:33:51.903",
    "logLevel": "DEBUG",
    "traceId": "180532b7-0cc7-057b-687a-5ca1824838f5",
    "status": "Success",
    "eventType": "FunctionExecution",
    "clientId": "N/A",
    "topicName":"rules/test",
    "ruleName": "ruleTestPredict",
    "ruleAction": "MachinelearningPredict",
```

```
    "resources": {
        "ModelId": "predict-model"
    },
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```

In addition to the Common CloudWatch Logs attributes ,`FunctionExecution` log entries contain the following attributes:

clientId

`N/A` for `FunctionExecution` logs.

principalId

The ID of the principal making the request.

resources

A collection of resources used by the rule's actions.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

RuleExecution log entry

When the AWS IoT rules engine triggers a rule's action, it generates a `RuleExecution` log entry.

RuleExecution log entry example

```
{
    "timestamp": "2017-08-10 16:32:46.070",
    "logLevel": "INFO",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "RuleExecution",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "rules/test",
    "ruleName": "JSONLogsRule",
```

```
    "ruleAction": "RepublishAction",
    "resources": {
        "RepublishTopic": "rules/republish"
    },
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```


RuleExecution with batching log entry example


```
{
    "logLevel": "INFO",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "RuleExecution",
    "ruleName": "rule_test",
    "ruleAction": "HttpAction",
    "resources": {
        "Url": "https://example.com",
        "ConfirmationUrl": "https://example.com"
    },
    "details": "HttpAction made a request to the specified endpoint",
    "batchDetails": {
        "timestamps": [
            "1234567890123",
            "1234567890123",
            "1234567890123"
        ],
        "traceIds": [
            "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
            "30aa7ccc-1d23-0b97-aa7b-76196d83537c",
            "30aa7ccc-1d23-0b97-aa7b-76196d83537d"
        ],
        "clientIds": [
            "N/A",
            "N/A",
            "N/A"
        ],
        "topicNames": [
            "topic/ruletest",
            "topic/ruletest",
            "topic/ruletest"
        ],
        "principalIds": [
```

```
            "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
            "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
            "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
        ],
        "batchSize": 3,
        "batchSizeInBytes": 114
    }
}
```

In addition to the Common CloudWatch Logs attributes, `RuleExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

resources

A collection of resources used by the rule's actions.

ruleAction

The name of the action triggered.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

RuleMatch log entry

The AWS IoT rules engine generates a log entry with an `eventType` of `RuleMatch` when the message broker receives a message that matches a rule.

RuleMatch log entry example

```
{
    "timestamp": "2017-08-10 16:32:46.002",
    "logLevel": "INFO",
```

```
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "RuleMatch",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "rules/test",
    "ruleName": "JSONLogsRule",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```

In addition to the Common CloudWatch Logs attributes ,`RuleMatch` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

RuleExecutionThrottled log entry

When an execution is throttled, the AWS IoT rules engine generates a log entry with an `eventType` of `RuleExecutionThrottled`.

RuleExecutionThrottled log entry example

```
{
    "timestamp": "2017-10-04 19:25:46.070",
    "logLevel": "ERROR",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "RuleExecutionThrottled",
    "clientId": "abf27092886e49a8a5c1922749736453",
```

```
    "topicName": "$aws/rules/example_rule",
    "ruleName": "example_rule",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "reason": "RuleExecutionThrottled",
    "details": "Exection of Rule example_rule throttled"
}
```


In addition to the Common CloudWatch Logs attributes ,`RuleExecutionThrottled` log entries contain the following attributes:

clientId

The ID of the client making the request.

details

A brief explanation of the error.

principalId

The ID of the principal making the request.

reason

The string "RuleExecutionThrottled".

ruleName

The name of the rule to be triggered.

topicName

The name of the topic that was published.

RuleNotFound log entry

When the AWS IoT rules engine cannot find a rule with a given name, it generates a log entry with an `eventType` of `RuleNotFound`.

RuleNotFound log entry example


```
{
    "timestamp": "2017-10-04 19:25:46.070",
    "logLevel": "ERROR",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
```

```
    "status": "Failure",
    "eventType": "RuleNotFound",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "$aws/rules/example_rule",
    "ruleName": "example_rule",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167",
    "reason": "RuleNotFound",
    "details": "Rule example_rule not found"
}
```

In addition to the Common CloudWatch Logs attributes ,`RuleNotFound` log entries contain the following attributes:

clientId

The ID of the client making the request.

details

A brief explanation of the error.

principalId

The ID of the principal making the request.

reason

The string "RuleNotFound".

ruleName

The name of the rule that could not be found.

topicName

The name of the topic that was published.

StartingRuleExecution log entry

When the AWS IoT rules engine starts to trigger a rule's action, it generates a log entry with an `eventType` of `StartingRuleExecution`.

StartingRuleExecution log entry example

```
{
```

```
    "timestamp": "2017-08-10 16:32:46.002",
    "logLevel": "DEBUG",
    "traceId": "30aa7ccc-1d23-0b97-aa7b-76196d83537e",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "StartingRuleExecution",
    "clientId": "abf27092886e49a8a5c1922749736453",
    "topicName": "rules/test",
    "ruleName": "JSONLogsRule",
    "ruleAction": "RepublishAction",
    "principalId": "145179c40e2219e18a909d896a5340b74cf97a39641beec2fc3eeafc5a932167"
}
```

In addition to the Common CloudWatch Logs attributes ,`rule-` log entries contain the following attributes:

clientId

The ID of the client making the request.

principalId

The ID of the principal making the request.

ruleAction

The name of the action triggered.

ruleName

The name of the matching rule.

topicName

The name of the subscribed topic.

Job log entries

The AWS IoT Job service generates log entries for the following events. Log entries are generated when an MQTT or HTTP request is received from the device.

Topics

- DescribeJobExecution log entry

- GetPendingJobExecution log entry
- ReportFinalJobExecutionCount log entry
- StartNextPendingJobExecution log entry
- UpdateJobExecution log entry

DescribeJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `DescribeJobExecution` when the service receives a request to describe a job execution.

DescribeJobExecution log entry example


```
{
    "timestamp": "2017-08-10 19:13:22.841",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "DescribeJobExecution",
    "protocol": "MQTT",
    "clientId": "thingOne",
    "jobId": "002",
    "topicName": "$aws/things/thingOne/jobs/002/get",
    "clientToken": "myToken",
    "details": "The request status is SUCCESS."
}
```


In addition to the Common CloudWatch Logs attributes ,`GetJobExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

clientToken

A unique, case-sensitive identifier to ensure the idempotency of the request. For more information, see How to Ensure Idempotency.

details

Other information from the Jobs service.

jobId

The job ID for the job execution.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The topic used to make the request.

GetPendingJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `GetPendingJobExecution` when the service receives a job execution request.

GetPendingJobExecution log entry example

```
{
    "timestamp": "2018-06-13 17:45:17.197",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "GetPendingJobExecution",
    "protocol": "MQTT",
    "clientId": "299966ad-54de-40b4-99d3-4fc8b52da0c5",
    "topicName": "$aws/things/299966ad-54de-40b4-99d3-4fc8b52da0c5/jobs/get",
    "clientToken": "24b9a741-15a7-44fc-bd3c-1ff2e34e5e82",
    "details": "The request status is SUCCESS."
}
```

In addition to the Common CloudWatch Logs attributes ,`GetPendingJobExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

clientToken

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see How to Ensure Idempotency.

details

Other information from the Jobs service.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The name of the subscribed topic.

ReportFinalJobExecutionCount log entry

The AWS IoT Jobs service generates a log entry with an `entryType` of `ReportFinalJobExecutionCount` when a job is completed.

ReportFinalJobExecutionCount log entry example


```
{
    "timestamp": "2017-08-10 19:44:16.776",
    "logLevel": "INFO",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "ReportFinalJobExecutionCount",
    "jobId": "002",
    "details": "Job 002 completed. QUEUED job execution count: 0 IN_PROGRESS job
 execution count: 0 FAILED job execution count: 0 SUCCEEDED job execution count: 1
 CANCELED job execution count: 0 REJECTED job execution count: 0 REMOVED job execution
 count: 0"
}
```


In addition to the [Common CloudWatch Logs attributes](#) ,`ReportFinalJobExecutionCount` log entries contain the following attributes:

details

Other information from the Jobs service.

jobId

The job ID for the job execution.

StartNextPendingJobExecution log entry

When it receives a request to start the next pending job execution, the AWS IoT Jobs service generates a log entry with an `eventType` of `StartNextPendingJobExecution`.

StartNextPendingJobExecution log entry example

```
{
    "timestamp": "2018-06-13 17:49:51.036",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "StartNextPendingJobExecution",
    "protocol": "MQTT",
    "clientId": "95c47808-b1ca-4794-bc68-a588d6d9216c",
    "topicName": "$aws/things/95c47808-b1ca-4794-bc68-a588d6d9216c/jobs/start-next",
    "clientToken": "bd7447c4-3a05-49f4-8517-dd89b2c68d94",
    "details": "The request status is SUCCESS."
}
```

In addition to the Common CloudWatch Logs attributes ,`StartNextPendingJobExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

clientToken

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see How to Ensure Idempotency.

details

Other information from the Jobs service.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The topic used to make the request.

UpdateJobExecution log entry

The AWS IoT Jobs service generates a log entry with an `eventType` of `UpdateJobExecution` when the service receives a request to update a job execution.

UpdateJobExecution log entry example

```
{
    "timestamp": "2017-08-10 19:25:14.758",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "eventType": "UpdateJobExecution",
    "protocol": "MQTT",
    "clientId": "thingOne",
    "jobId": "002",
    "topicName": "$aws/things/thingOne/jobs/002/update",
    "clientToken": "myClientToken",
    "versionNumber": "1",
    "details": "The destination status is IN_PROGRESS. The request status is SUCCESS."
}
```

In addition to the Common CloudWatch Logs attributes ,`UpdateJobExecution` log entries contain the following attributes:

clientId

The ID of the client making the request.

clientToken

A unique, case sensitive identifier to ensure the idempotency of the request. For more information, see How to Ensure Idempotency.

details

Other information from the Jobs service.

jobId

The job ID for the job execution.

protocol

The protocol used to make the request. Valid values are `MQTT` or `HTTP`.

topicName

The topic used to make the request.

versionNumber

The version of the job execution.

Device provisioning log entries

The AWS IoT Device Provisioning service generates logs for the following events.

Topics

GetDeviceCredentials log entry

The AWS IoT Device Provisioning service generates a log entry with an `eventType` of `GetDeviceCredential` when a client calls `GetDeviceCredential`.

GetDeviceCredentials log entry example


```
{
  "timestamp" : "2019-02-20 20:31:22.932",
  "logLevel" : "INFO",
  "traceId" : "8d9c016f-6cc7-441e-8909-7ee3d5563405",
  "accountId" : "123456789101",
  "status" : "Success",
  "eventType" : "GetDeviceCredentials",
  "deviceCertificateId" :
 "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "details" : "Additional details about this log."
}
```


In addition to the Common CloudWatch Logs attributes ,`GetDeviceCredentials` log entries contain the following attributes:

details

A brief explanation of the error.

deviceCertificateId

The ID of the device certificate.

ProvisionDevice log entry

The AWS IoT Device Provisioning service generates a log entry with an `eventType` of `ProvisionDevice` when a client calls `ProvisionDevice`.

ProvisionDevice log entry example

```
{
  "timestamp" : "2019-02-20 20:31:22.932",
  "logLevel" : "INFO",
  "traceId" : "8d9c016f-6cc7-441e-8909-7ee3d5563405",
  "accountId" : "123456789101",
  "status" : "Success",
  "eventType" : "ProvisionDevice",
  "provisioningTemplateName" : "myTemplate",
  "deviceCertificateId" :
 "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "details" : "Additional details about this log."
 }
```

In addition to the [Common CloudWatch Logs attributes](#) ,`ProvisionDevice` log entries contain the following attributes:

details

A brief explanation of the error.

deviceCertificateId

The ID of the device certificate.

provisioningTemplateName

The name of the provisioning template.

Dynamic thing group log entries

AWS IoT Dynamic Thing Groups generate logs for the following event.

Topics

- [AddThingToDynamicThingGroupsFailed log entry]

AddThingToDynamicThingGroupsFailed log entry

When AWS IoT was not able to add a thing to the specified dynamic groups, it generates a log entry with an `eventType` of `AddThingToDynamicThingGroupsFailed`. This happens when a thing met the criteria to be in the dynamic thing group; however, it could not be added to the dynamic group or it was removed from the dynamic group. This can happen because:

- The thing already belongs to the maximum number of groups.
- The **--override-dynamic-groups** option was used to add the thing to a static thing group. It was removed from a dynamic thing group to make that possible.

For more information, see Dynamic Thing Group Limitations and Conflicts.

AddThingToDynamicThingGroupsFailed log entry example

This example shows the log entry of an `AddThingToDynamicThingGroupsFailed` error. In this example, *TestThing* met the criteria to be in the dynamic thing groups listed in `dynamicThingGroupNames`, but could not be added to those dynamic groups, as described in `reason`.

```
{
 "timestamp": "2020-03-16 22:24:43.804",
 "logLevel": "ERROR",
 "traceId": "70b1f2f5-d95e-f897-9dcc-31e68c3e1a30",
 "accountId": "57EXAMPLE833",
 "status": "Failure",
 "eventType": "AddThingToDynamicThingGroupsFailed",
 "thingName": "TestThing",
 "dynamicThingGroupNames": [
  "DynamicThingGroup11",
  "DynamicThingGroup12",
  "DynamicThingGroup13",
  "DynamicThingGroup14"
 ],
 "reason": "The thing failed to be added to the given dynamic thing group(s) because
 the thing already belongs to the maximum allowed number of groups."
}
```

In addition to the Common CloudWatch Logs attributes , AddThingToDynamicThingGroupsFailed log entries contain the following attributes:

dynamicThingGroupNames

An array of the dynamic thing groups to which the thing could not be added.

reason

The reason why the thing could not be added to the dynamic thing groups.

thingName

The name of the thing that could not be added to a dynamic thing group.

Fleet indexing log entries

AWS IoT fleet indexing generates log entries for the following events.

Topics

- NamedShadowCountForDynamicGroupQueryLimitExceeded log entry

NamedShadowCountForDynamicGroupQueryLimitExceeded log entry

A maximum of 25 named shadows per thing are processed for query terms that are not data source specific in dynamic groups. When this limit is breached for a thing, the NamedShadowCountForDynamicGroupQueryLimitExceeded event type will be emitted.

NamedShadowCountForDynamicGroupQueryLimitExceeded log entry example

This example shows the log entry of a NamedShadowCountForDynamicGroupQueryLimitExceeded error. In this example, all-values based DynamicGroup results can be inaccurate, as described in the reason field.

```
{
"timestamp": "2020-03-16 22:24:43.804",
"logLevel": "ERROR",
"traceId": "70b1f2f5-d95e-f897-9dcc-31e68c3e1a30",
"accountId": "571032923833",
"status": "Failure",
"eventType": "NamedShadowCountForDynamicGroupQueryLimitExceeded",
```

```
"thingName": "TestThing",
"reason": "A maximum of 25 named shadows per thing are processed for non-data source
 specific query terms in dynamic groups."
}
```

Common CloudWatch Logs attributes

All CloudWatch Logs log entries include these attributes:

accountId

Your AWS account ID.

eventType

The event type for which the log was generated. The value of the event type depends on the event that generated the log entry. Each log entry description includes the value of `eventType` for that log entry.

logLevel

The log level being used. For more information, see the section called “Log levels” .

status

The status of the request.

timestamp

The human-readable UTC timestamp of when the client connected to the AWS IoT message broker.

traceId

A randomly generated identifier that can be used to correlate all logs for a specific request.

Upload device-side logs to Amazon CloudWatch

You can upload historical, device-side logs into Amazon CloudWatch to monitor and analyze a device's activity in the field. Device-side logs can include system, application, and device logs files. This process uses a CloudWatch Logs rules action parameter to publish device-side logs into a customer-defined log group.

How it works

The process begins when an AWS IoT device sends MQTT messages containing formatted log files to an AWS IoT topic. An AWS IoT rule monitors the message topic and sends the log files to a CloudWatch Logs group that you define. You can then review and analyze the information.

Topics

- MQTT topics
- Rule action

MQTT topics

Choose an MQTT topic name space that you will use to publish the logs. We recommend using this format for the common topic space, `$aws/rules/things/thing_name/logs`, and this format for error topics, `$aws/rules/things/thing_name/logs/errors`. The naming structure for logs and error topics is recommended, but not required. For more information, see Designing MQTT Topics for AWS IoT Core.

By using the recommended common topic space, you utilize AWS IoT Basic Ingest reserved topics. AWS IoT Basic Ingest securely sends device data to the AWS services that are supported by AWS IoT rule actions. It removes the publish/subscribe message broker from the ingestion path, making it more cost effective. For more information, see Reducing messaging costs with Basic Ingest.

If you use batchMode to upload log files, your messages must follow a specific format that includes a UNIX timestamp and message. For more information, see the MQTT message format requirements for batchMode topic within CloudWatch Logs rule action.

Rule action

When AWS IoT receives the MQTT messages from the client devices, an AWS IoT rule monitors the customer-defined topic and publishes the contents into a CloudWatch log group that you define. This process uses a CloudWatch Logs rule action to monitor MQTT for batches of log files. For more information, see the CloudWatch Logs AWS IoT rule action.

Batch mode

`batchMode` is a Boolean parameter within the AWS IoT CloudWatch Logs rule action. This parameter is optional and is off (`false`) by default. To upload device-side log files in batches, you must turn this parameter on (`true`) when you create the AWS IoT rule. For more information, see CloudWatch Logs in the AWS IoT rule actions section.

Uploading device-side logs by using AWS IoT rules

You can use the AWS IoT rules engine to upload log records from existing device-side log files (system, application, and device-client logs) to Amazon CloudWatch. When device-side logs are published to an MQTT topic, the CloudWatch Logs rules action transfers the messages to CloudWatch Logs. This process outlines how to upload device logs in batches using the rules action `batchMode` parameter turned on (set to `true`).

To begin uploading device-side logs to CloudWatch, complete the following prerequisites.

Prerequisites

Before you begin, do the following:

- Create at least one target IoT device that's registered with AWS IoT Core as an AWS IoT thing. For more information, see Create a thing object.
- Determine the MQTT topic space for ingestion and errors. For more information about MQTT topics and recommended naming conventions, see the MQTT topics MQTT topics section in Upload device-side logs to Amazon CloudWatch.

For more information about these prerequisites, see Upload device-side logs to CloudWatch.

Creating a CloudWatch log group

To create a CloudWatch log group, complete the following steps. Choose the appropriate tab depending on whether you prefer to perform the steps through the AWS Management Console or the AWS Command Line Interface (AWS CLI).

AWS Management Console

To create a CloudWatch log group by using the AWS Management Console

1. Open the AWS Management Console and navigate to CloudWatch.
2. On the navigation bar, choose **Logs**, and then **Log groups**.
3. Choose **Create log group**.

4. Update the **Log group name** and, optionally, update the **Retention setting** fields.
5. Choose **Create**.

AWS CLI

To create a CloudWatch log group by using the AWS CLI

1. To create the log group, run the following command. For more information, see create-log-group in the AWS CLI v2 Command Reference.

   Replace the log group name in the example (uploadLogsGroup) with your preferred name.

```
aws logs create-log-group --log-group-name uploadLogsGroup
```

2. To confirm that the log group was created correctly, run the following command.

```
aws logs describe-log-groups --log-group-name-prefix uploadLogsGroup
```

   Sample output:

```
{
    "logGroups": [
        {
            "logGroupName": "uploadLogsGroup",
            "creationTime": 1674521804657,
            "metricFilterCount": 0,
            "arn": "arn:aws:logs:us-east-1:111122223333:log-
group:uploadLogsGroup:*",
            "storedBytes": 0
        }
    ]
}
```

Creating a topic rule

To create an AWS IoT rule, complete the following steps. Choose the appropriate tab depending on whether you prefer to perform the steps through the AWS Management Console or the AWS Command Line Interface (AWS CLI).

AWS Management Console

To create a topic rule by using the AWS Management Console

1. Open the Rule hub.

   a. Open the AWS Management Console and navigate to AWS IoT .

   b. On the navigation bar, choose **Message routing** and then **Rules**.

   c. Choose **Create rule**.

2. Enter the rule properties.

   a. Enter an alphanumeric **Rule name**.

   b. (Optional) Enter a **Rule description** and **Tags**.

   c. Choose **Next**.

3. Enter a SQL statement.

   a. Enter a SQL statement using the MQTT topic that you defined for ingestion.

      For example, `SELECT * FROM '$aws/rules/things/thing_name/logs'`

   b. Choose **Next**.

4. Enter rule actions.

   a. On the **Action 1** menu, choose **CloudWatch logs**.

   b. Choose the **Log group name** and then choose the log group that you created.

   c. Select **Use batch mode**.

   d. Specify the IAM role for the rule.

      If you have an IAM role for the rule, do the following.

      1. On the **IAM role** menu, choose your IAM role.

      If you don't have an IAM role for the rule, do the following.

      1. Choose **Create new role**.
      2. For **Role name**, enter a unique name and choose **Create**.
      3. Confirm that the IAM role name is correct in the **IAM role** field.

   e. Choose **Next**.

5. Review the template configuration.

   a. Review the settings for the Job template to verify they're correct.

   b. When you're done, choose **Create**.

AWS CLI

To create an IAM role and a topic rule by using the AWS CLI

1. Create an IAM role that grants rights to the AWS IoT rule.

   a. Create an IAM policy.

      To create an IAM policy, run the following command. Make sure you update the `policy-name` parameter value. For more information, see [`create-policy`](create-policy) in the AWS CLI v2 Command Reference.

      > **Note**
      >
      > If you're using a Microsoft Windows operating system, you might need to replace the end of line marker (\) with a tick (`) or another character.

```
aws iam create-policy \
    --policy-name uploadLogsPolicy \
    --policy-document \
'{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "iot:CreateTopicRule",
            "iot:Publish",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:GetLogEvents"
        ],
        "Resource": "*"
    }
```

```
}'
```

b. Copy the policy ARN from your output into a text editor.

Sample output:

```
{
    "Policy": {
        "PolicyName": "uploadLogsPolicy",
        "PermissionsBoundaryUsageCount": 0,
        "CreateDate": "2023-01-23T18:30:10Z",
        "AttachmentCount": 0,
        "IsAttachable": true,
        "PolicyId": "AAABBBCCCDDDEEEFFFGGG",
        "DefaultVersionId": "v1",
        "Path": "/",
        "Arn": "arn:aws:iam::111122223333:policy/uploadLogsPolicy",
        "UpdateDate": "2023-01-23T18:30:10Z"
    }
}
```

c. Create an IAM role and trust policy.

To create an IAM policy, run the following command. Make sure you update the `role-name` parameter value. For more information, see create-role in the AWS CLI v2 Command Reference.

```
aws iam create-role \
--role-name uploadLogsRole \
--assume-role-policy-document \
'{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
```

```
}'
```

d. Attach the IAM policy to the rule.

   To create an IAM policy, run the following command. Make sure you update the `role-name` and `policy-arn` parameter values. For more information, see attach-role-policy in the AWS CLI v2 Command Reference.

```
aws iam attach-role-policy \
--role-name uploadLogsRole \
--policy-arn arn:aws:iam::111122223333:policy/uploadLogsPolicy
```

e. Review the role.

   To confirm that the IAM role was created correctly, run the following command. Make sure you update the `role-name` parameter value. For more information, see get-role in the AWS CLI v2 Command Reference.

```
aws iam get-role --role-name uploadLogsRole
```

   Sample output:

```
{
    "Role": {
        "Path": "/",
        "RoleName": "uploadLogsRole",
        "RoleId": "AAABBBCCCDDDEEEFFFGGG",
        "Arn": "arn:aws:iam::111122223333:role/uploadLogsRole",
        "CreateDate": "2023-01-23T19:17:15+00:00",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Sid": "Statement1",
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "iot.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        },
```

```
        "Description": "",
        "MaxSessionDuration": 3600,
        "RoleLastUsed": {}
    }
}
```

2. Create an AWS IoT topic rule in the AWS CLI.

   a. To create an AWS IoT topic rule, run the following command. Make sure you update the `--rule-name`,`sql` statement, `description`,`roleARN` , and `logGroupName` parameter values. For more information, see create-topic-rule in the AWS CLI v2 Command Reference.

```
aws iot create-topic-rule \
--rule-name uploadLogsRule \
--topic-rule-payload \
 '{
  "sql":"SELECT * FROM 'rules/things/thing_name/logs'",
  "description":"Upload logs test rule",
  "ruleDisabled":false,
  "awsIotSqlVersion":"2016-03-23",
  "actions":[
   {"cloudwatchLogs":
    {"roleArn":"arn:aws:iam::111122223333:role/uploadLogsRole",
    "logGroupName":"uploadLogsGroup",
    "batchMode":true}
    }
   ]
}'
```

   b. To confirm that the rule was created correctly, run the following command. Make sure you update the `role-name` parameter value. For more information, see get-topic-rule in the AWS CLI v2 Command Reference.

```
aws iot get-topic-rule --rule-name uploadLogsRule
```

   Sample output:

```
{
    "ruleArn": "arn:aws:iot:us-east-1:111122223333:rule/uploadLogsRule",
    "rule": {
        "ruleName": "uploadLogsRule",
```

```
        "sql": "SELECT * FROM rules/things/thing_name/logs",
        "description": "Upload logs test rule",
        "createdAt": "2023-01-24T16:28:15+00:00",
        "actions": [
            {
                "cloudwatchLogs": {
                    "roleArn": "arn:aws:iam::111122223333:role/
uploadLogsRole",
                    "logGroupName": "uploadLogsGroup",
                    "batchMode": true
                }
            }
        ],
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23"
    }
}
```

Sending device-side logs to AWS IoT

To send device-side logs to AWS IoT

1. To send historical logs to AWS IoT, communicate with your devices to ensure the following.

   - The log information is sent to the correct topic namespace as specified within the *Prerequisites* section of this procedure.

     For example, `$aws/rules/things/thing_name/logs`
   - The MQTT message payload is formatted correctly. For more information about MQTT topic and recommended naming convention, see the MQTT topics section within Upload device-side logs to Amazon CloudWatch .
2. Confirm that the MQTT messages are received within the AWS IoT MQTT client.

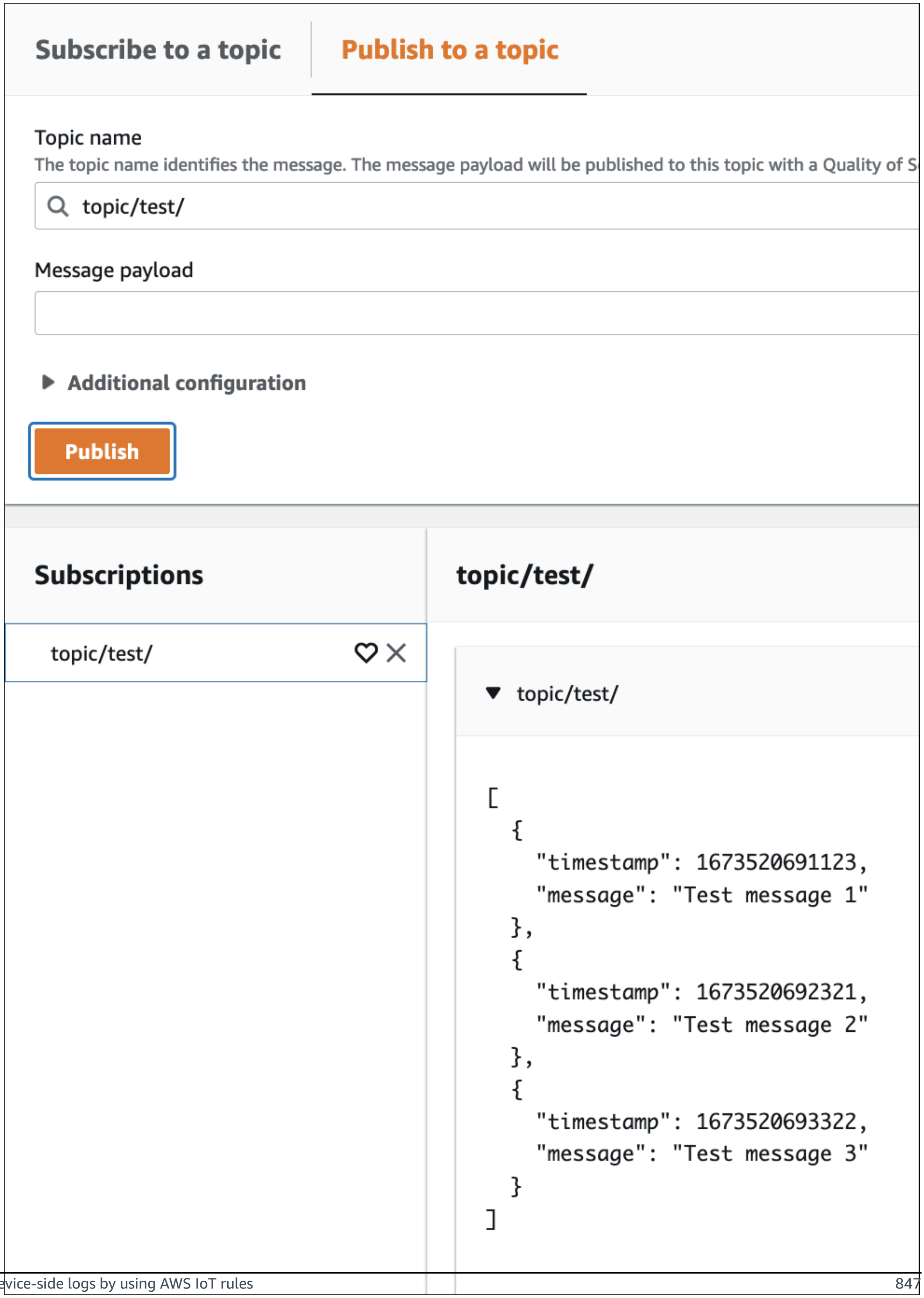
   a. Open the AWS Management Console and navigate to AWS IoT.

   b. To view the **MQTT test client**, on the navigation bar, choose **Test**,**MQTT test client**.

   c. For **Subscribe to a topic**,**Topic filter**, enter the *topic namespace*.

   d. Choose **Subscribe**.

MQTT messages appear in the **Subscriptions** and **Topic** table, as seen in the following. These messages can take up to five minutes to appear.


Subscribe to a topic
Publish to a topic
Topic name
The topic name identifies the message. The message payload will be published to this topic with a Quality of S
topic/test/
Message payload
Additional configuration
Publish
Subscriptions
topic/test/
topic/test/
topic/test/
[
{
"timestamp": 1673520691123,
"message": "Test message 1"
},
{
"timestamp": 1673520692321,
"message": "Test message 2"
},
{
"timestamp": 1673520693322,
"message": "Test message 3"
}
]

Viewing the log data

To review your log records in CloudWatch Logs

1. Open the AWS Management Console, and navigate to CloudWatch.
2. On the navigation bar, choose **Logs**,**Logs Insights**.
3. On the **Select log group(s)** menu, choose the log group you specified in the AWS IoT rule.
4. On the **Logs insights** page, choose **Run query**.

Logging AWS IoT API calls using AWS CloudTrail

AWS IoT is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS IoT. CloudTrail captures all API calls for AWS IoT as events, including calls from the AWS IoT console and from code calls to the AWS IoT APIs. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS IoT. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT, the IP address from which the request was made, who made the request, when it was made, and other details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS IoT information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for AWS IoT, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all AWS Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. You can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations

- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

> **Note**
>
> AWS IoT data plane actions (device side) are not logged by CloudTrail. Use CloudWatch to monitor these actions.

Generally speaking, AWS IoT control plane actions that make changes are logged by CloudTrail. Calls such as **CreateThing**, **CreateKeysAndCertificate**, and **UpdateCertificate** leave CloudTrail entries, while calls such as **ListThings** and **ListTopicRules** do not.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

AWS IoT actions are documented in the AWS IoT API Reference. AWS IoT Wireless actions are documented in the AWS IoT Wireless API Reference.

Understanding AWS IoT log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `AttachPolicy` action.

```
{
```

```
    "timestamp":"1460159496",
    "AdditionalEventData":"",
    "Annotation":"",
    "ApiVersion":"",
    "ErrorCode":"",
    "ErrorMessage":"",
    "EventID":"8bff4fed-c229-4d2d-8264-4ab28a487505",
    "EventName":"AttachPolicy",
    "EventTime":"2016-04-08T23:51:36Z",
    "EventType":"AwsApiCall",
    "ReadOnly":"",
    "RecipientAccountList":"",
    "RequestID":"d4875df2-fde4-11e5-b829-23bf9b56cbcd",
    "RequestParamters":{
        "principal":"arn:aws:iot:us-
east-1:123456789012:cert/528ce36e8047f6a75ee51ab7beddb4eb268ad41d2ea881a10b67e8e76924d894",
        "policyName":"ExamplePolicyForIoT"
    },
    "Resources":"",
    "ResponseElements":"",
    "SourceIpAddress":"52.90.213.26",
    "UserAgent":"aws-internal/3",
    "UserIdentity":{
        "type":"AssumedRole",
        "principalId":"AKIAI44QH8DHBEXAMPLE",
        "arn":"arn:aws:sts::12345678912:assumed-role/iotmonitor-us-east-1-beta-
InstanceRole-1C5T1YCYMHPYT/i-35d0a4b6",
        "accountId":"222222222222",
        "accessKeyId":"access-key-id",
        "sessionContext":{
            "attributes":{
                "mfaAuthenticated":"false",
                "creationDate":"Fri Apr 08 23:51:10 UTC 2016"
            },
            "sessionIssuer":{
                "type":"Role",
                "principalId":"AKIAI44QH8DHBEXAMPLE",
                "arn":"arn:aws:iam::123456789012:role/executionServiceEC2Role/
iotmonitor-us-east-1-beta-InstanceRole-1C5T1YCYMHPYT",
                "accountId":"222222222222",
                "userName":"iotmonitor-us-east-1-InstanceRole-1C5T1YCYMHPYT"
            }
        },
        "invokedBy":{
```

```
            "serviceAccountId":"111111111111"
        }
    },
    "VpcEndpointId":""
}
```

Rules for AWS IoT

Rules give your devices the ability to interact with AWS services. Rules are analyzed and actions are performed based on the MQTT topic stream. You can use rules to support the following tasks:

- Augment or filter data received from a device.
- Write data received from a device to an Amazon DynamoDB database.
- Save a file to Amazon S3.
- Send a push notification to all users who are using Amazon SNS.
- Publish data to an Amazon SQS queue.
- Invoke a Lambda function to extract data.
- Process messages from a large number of devices using Amazon Kinesis.
- Send data to Amazon OpenSearch Service.
- Capture a CloudWatch metric.
- Change a CloudWatch alarm.
- Send the data from an MQTT message to Amazon SageMaker AI to make predictions based on a machine learning (ML) model.
- Send a message to a Salesforce IoT Input Stream.
- Start process of a Step Functions state machine.
- Send message data to an AWS IoT Events input.
- Send message data to an asset property in AWS IoT SiteWise.
- Send message data to a web application or service.

Your rules can use MQTT messages that pass through the publish/subscribe protocol supported by the the section called “Device communication protocols”. You can also use the Basic Ingest feature to securely send device data to the AWS services listed previously, without incurring messaging costs. The Basic Ingest feature optimizes data flow by removing the publish/subscribe message broker from the ingestion path. This makes it cost effective while still keeping the security and data processing features of AWS IoT.

Before AWS IoT can perform these actions, you must grant it permission to access your AWS resources on your behalf. When the actions are performed, you incur the standard charges for the AWS services that you use.

Contents

Granting an AWS IoT rule the access it requires

Use IAM roles to control the AWS resources to which each rule has access. Before you create a rule, you must create an IAM role with a policy that allows access to the required AWS resources. AWS IoT assumes this role when implementing a rule.

Complete the following steps to create the IAM role and AWS IoT policy that grant an AWS IoT rule the access it requires (AWS CLI).

1. Save the following trust policy document, which grants AWS IoT permission to assume the role, to a file named `iot-role-trust.json`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
        },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "123456789012"
```

```
            },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:iot:us-east-1:123456789012:rule/
rulename"
            }
        }
        }
    ]
}
```

Use the [create-role](https://docs.aws.amazon.com/cli/latest/reference/iam/create-role.html) command to create an IAM role specifying the `iot-role-trust.json` file:

```
aws iam create-role --role-name my-iot-role --assume-role-policy-document
 file://iot-role-trust.json
```

The output of this command looks like the following:

```
{
 "Role": {
  "AssumeRolePolicyDocument": "url-encoded-json",
  "RoleId": "AKIAIOSFODNN7EXAMPLE",
  "CreateDate": "2015-09-30T18:43:32.821Z",
  "RoleName": "my-iot-role",
  "Path": "/",
  "Arn": "arn:aws:iam::123456789012:role/my-iot-role"
 }
}
```

2. Save the following JSON into a file named `my-iot-policy.json`.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": "dynamodb:*",
   "Resource": "*"
  }
 ]
```

```
}
```

This JSON is an example policy document that grants AWS IoT administrator access to DynamoDB.

Use the create-policy command to grant AWS IoT access to your AWS resources upon assuming the role, passing in the `my-iot-policy.json` file:

```
aws iam create-policy --policy-name my-iot-policy --policy-document file://my-iot-
policy.json
```

For more information about how to grant access to AWS services in policies for AWS IoT, see Creating an AWS IoT rule.

The output of the create-policy command contains the ARN of the policy. Attach the policy to a role.

```
{
 "Policy": {
  "PolicyName": "my-iot-policy",
  "CreateDate": "2015-09-30T19:31:18.620Z",
  "AttachmentCount": 0,
  "IsAttachable": true,
  "PolicyId": "ZXR6A36LTYANPAI7NJ5UV",
  "DefaultVersionId": "v1",
  "Path": "/",
  "Arn": "arn:aws:iam::123456789012:policy/my-iot-policy",
  "UpdateDate": "2015-09-30T19:31:18.620Z"
 }
}
```

3. Use the attach-role-policy command to attach your policy to your role:

```
aws iam attach-role-policy --role-name my-iot-role --policy-arn
 "arn:aws:iam::123456789012:policy/my-iot-policy"
```

Revoke rule engine access

To immediately revoke rule engine access, do the following

1. Remove iot.amazonaws.com from the trust policy
2. Follow the steps to revoke iot role sessions

Passing role permissions

Part of a rule definition is an IAM role that grants permission to access resources specified in the rule's action. The rules engine assumes that role when the rule's action is invoked. The role must be defined in the same AWS account as the rule.

When creating or replacing a rule you are, in effect, passing a role to the rules engine. The `iam:PassRole` permission is required to perform this operation. To verify that you have this permission, create a policy that grants the `iam:PassRole` permission and attach it to your IAM user. The following policy shows how to allow `iam:PassRole` permission for a role.

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Sid": "Stmt1",
   "Effect": "Allow",
   "Action": [
    "iam:PassRole"
   ],
   "Resource": [
    "arn:aws:iam::123456789012:role/myRole"
   ]
  }
 ]
}
```

In this policy example, the `iam:PassRole` permission is granted for the role `myRole`. The role is specified using the role's ARN. Attach this policy to your IAM user or role that your user belongs to. For more information, see Working with Managed Policies.

> **Note**
>
> Lambda functions use resource-based policy, where the policy is attached directly to the Lambda function itself. When you create a rule that invokes a Lambda function, you don't

pass a role, so the user creating the rule doesn't need the `iam:PassRole` permission. For more information about Lambda function authorization, see Granting Permissions Using a Resource Policy.

Creating an AWS IoT rule

You can create AWS IoT rules to route data from your connected things to interact with other AWS services. An AWS IoT rule consists of the following components:

Components of a rule

Component	Description	Required or Optional
Rule name	The name of the rule. Note that we do not recommend the use of personally identifiable information in your rule names.	Required.
Rule description	A textual description of the rule. Note that we do not recommend the use of personally identifiable information in your rule descriptions.	Optional.
SQL statement	A simplified SQL syntax to filter messages received on an MQTT topic and push the data elsewhere. For more information, see AWS IoT SQL reference.	Required.
SQL version	The version of the SQL rules engine to use when evaluating the rule. Although this property is optional, we strongly recommend that you specify the SQL version. The AWS IoT Core console sets this property to `2016-03-23` by default. If this property is not set, such as in an AWS CLI command or an CloudFormation template, `2015-10-08` is used. For more information, see SQL versions.	Required.

Component	Description	Required or Optional
One or more actions	The actions AWS IoT performs when enacting the rule. For example, you can insert data into a DynamoDB table, write data to an Amazon S3 bucket, publish to an Amazon SNS topic, or invoke a Lambda function.	Required.
An error action	The action AWS IoT performs when it's unable to perform a rule's action.	Optional.

Before you create an AWS IoT rule, you must create an IAM role with a policy that allows access to the required AWS resources. AWS IoT assumes this role when implementing a rule. For more information, see Granting an AWS IoT rule the access it requires and Passing role permissions.

When you create a rule, be aware of how much data you're publishing on topics. If you create rules that include a wildcard topic pattern, they might match a large percentage of your messages. If this is the case, you might need to increase the capacity of the AWS resources used by the target actions. We recommend avoiding wildcard topic patterns in republish rules to prevent duplicate processing and reduce costs.

> **Note**
>
> Creating and updating rules are administrator-level actions. Any user who has permission to create or update rules is able to access data processed by the rules.

Create a rule (Console)

To create a rule (AWS Management Console)

Use the AWS Management Console command to create a rule:

1. Open the AWS IoT console.
2. On the left navigation, choose **Message routing** from **Manage** section. Then choose **Rules**.
3. On the **Rules** page, choose **Create rule**.

4. On the **Specify rule properties** page, enter a name for your rule. **Rule description** and **Tags** are optional. Choose **Next**.
5. On the **Configure SQL statement** page, choose a SQL version and enter a SQL statement. An example SQL statement can be `SELECT temperature FROM 'iot/topic' WHERE temperature > 50`. For more information, see SQL versions and AWS IoT SQL reference.
6. On the **Attach rule actions** page, add rule actions to route data to other AWS services.

    1. In **Rule actions**, select a rule action from the drop down list. For example, you can choose **Kinesis Stream**. For more information about rule actions, see AWS IoT rule actions.
    2. Depending on the rule action you choose, enter related configuration details. For example, if you choose **Kinesis Stream**, you will need to choose or create a data stream resource, and optionally enter configuration details such as **Partition key**, which is used to group data by shard in a steam.
    3. In **IAM role**, choose or create a role to grant AWS IoT access to your endpoint. Note that AWS IoT will automatically create a policy with a prefix of `aws-iot-rule` under your IAM role selected. You can choose **View** to view your IAM role and the policy from the IAM console. **Error action** is optional. You can find more information in Error handling (error action). For more information about creating an IAM role for your rule, see Grant a rule the access it requires. Choose **Next**.
7. On the **Review and create** page, review all the configuration and make edits if needed. Choose **Create**.

After you create a rule successfully, you will see the rule listed on the **Rules** page. You can select a rule to open the **Details** page where you can view a rule, edit a rule, deactivate a rule, and delete a rule.

Create a rule (CLI)

To create a rule (AWS CLI)

Use the create-topic-rule command to create a rule:

```
aws iot create-topic-rule --rule-name myrule --topic-rule-payload file://myrule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified DynamoDB table. The SQL statement filters the messages and the role ARN grants AWS IoT permission to write to the DynamoDB table.

```
{
 "sql": "SELECT * FROM 'iot/test'",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "dynamoDB": {
    "tableName": "my-dynamodb-table",
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
    "hashKeyField": "topic",
    "hashKeyValue": "${topic(2)}",
    "rangeKeyField": "timestamp",
    "rangeKeyValue": "${timestamp()}"
   }
  }
 ]
}
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified S3 bucket. The SQL statement filters the messages, and the role ARN grants AWS IoT permission to write to the Amazon S3 bucket.

```
{
 "awsIotSqlVersion": "2016-03-23",
 "sql": "SELECT * FROM 'iot/test'",
 "ruleDisabled": false,
 "actions": [
  {
   "s3": {
    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",
    "bucketName": "amzn-s3-demo-bucket",
    "key": "myS3Key"
   }
  }
 ]
}
```

The following is an example payload file with a rule that pushes data to Amazon OpenSearch Service:

```
{
 "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",
```

```
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "OpenSearch": {
    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es",
    "endpoint": "https://my-endpoint",
    "index": "my-index",
    "type": "my-type",
    "id": "${newuuid()}"
   }
  }
 ]
}
```

The following is an example payload file with a rule that invokes a Lambda function:

```
{
 "sql": "expression",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "lambda": {
    "functionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function"
   }
  }
 ]
}
```

The following is an example payload file with a rule that publishes to an Amazon SNS topic:

```
{
 "sql": "expression",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "sns": {
    "targetArn": "arn:aws:sns:us-west-2:123456789012:my-sns-topic",
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
   }
  }
```

```
 ]
}
```

The following is an example payload file with a rule that republishes on a different MQTT topic:

```
{
 "sql": "expression",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "republish": {
    "topic": "my-mqtt-topic",
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
   }
  }
 ]
}
```

The following is an example payload file with a rule that pushes data to an Amazon Data Firehose stream:

```
{
 "sql": "SELECT * FROM 'my-topic'",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "firehose": {
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
    "deliveryStreamName": "my-stream-name"
   }
  }
 ]
}
```

The following is an example payload file with a rule that uses the Amazon SageMaker AI `machinelearning_predict` function to republish to a topic if the data in the MQTT payload is classified as a 1.

```
{
```

```
 "sql": "SELECT * FROM 'iot/test' where machinelearning_predict('my-model',
 'arn:aws:iam::123456789012:role/my-iot-aml-role', *).predictedLabel=1",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "republish": {
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
    "topic": "my-mqtt-topic"
   }
  }
 ]
}
```

The following is an example payload file with a rule that publishes messages to a Salesforce IoT Cloud input stream.

```
{
 "sql": "expression",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "salesforce": {
    "token": "ABCDEFGHI123456789abcdefghi123456789",
    "url": "https://ingestion-cluster-id.my-env.sfdcnow.com/streams/stream-id/
connection-id/my-event"
   }
  }
 ]
}
```

The following is an example payload file with a rule that starts an execution of a Step Functions state machine.

```
{
 "sql": "expression",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "stepFunctions": {
```

```
    "stateMachineName": "myCoolStateMachine",
    "executionNamePrefix": "coolRunning",
    "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
   }
  }
 ]
}
```

Managing an AWS IoT rule

You can use the following actions to manage your AWS IoT rules.

In this topic:

- Tagging a rule
- Viewing a rule
- Deleting a rule

Tagging a rule

To add another layer of specificity to your new or existing rules, you can apply tagging. Tagging leverages key-value pairs in your rules to provide you with greater control over how and where your rules are applied to your AWS IoT resources and services. For example, you can limit the scope of your rule to only apply in your beta environment for pre release testing (`Key=environment, Value=beta`) or capturing all messages sent to the `iot/test` topic from a specific endpoint only and storing them in an Amazon S3 bucket.

IAM policy example

For an example that shows how to grant tagging permissions for a rule, consider a user that runs the following command to create a rule and tag it to apply only to their beta environment.

In the example, replace:

- *`MyTopicRuleName`* with the name of the rule.
- *`myrule.json`* with the name of the policy document.

```
aws iot create-topic-rule
    --rule-name MyTopicRuleName
```

```
    --topic-rule-payload file://myrule.json
    --tags "environment=beta"
```

For this example, you must use the following IAM policy:

```
{
    "Version":"2012-10-17",
    "Statement":
    {
        "Action": [ "iot:CreateTopicRule", "iot:TagResource" ],
        "Effect": "Allow",
        "Resource": [
            "arn:aws:iot:us-east-1:123456789012:rule/MyTopicRuleName"
        ]
    }
}
```

The above example shows a newly created rule called `MyTopicRuleName` that applies only to your beta environment. The `iot:TagResource` in the policy statement with `MyTopicRuleName` specifically called out allows tagging when creating or updating `MyTopicRuleName`. The parameter `--tags "environment=beta"` used when creating the rule limits the scope of `MyTopicRuleName` to only your beta environment. If you remove the parameter `--tags "environment=beta"`, then `MyTopicRuleName` will apply to all environments.

For more information on creating IAM roles and policies specific to an AWS IoT rule, see Granting an AWS IoT rule the access it requires

For general information about tagging your resources, see Tagging your AWS IoT resources.

Viewing a rule

Use the list-topic-rules command to list your rules:

```
aws iot list-topic-rules
```

Use the get-topic-rule command to get information about a rule:

```
aws iot get-topic-rule --rule-name myrule
```

Deleting a rule

When you are finished with a rule, you can delete it.

To delete a rule (AWS CLI)

Use the delete-topic-rule command to delete a rule:

```
aws iot delete-topic-rule --rule-name myrule
```

AWS IoT rule actions

AWS IoT rule actions specify what to do when a rule is invoked. You can define actions to send data to an Amazon DynamoDB database, send data to Amazon Kinesis Data Streams, invoke an AWS Lambda function, and so on. AWS IoT supports the following actions in AWS Regions where the action's service is available.

Rule action	Description	Name in API
Apache Kafka	Sends a message to an Apache Kafka cluster.	`kafka`
CloudWatch alarms	Changes the state of an Amazon CloudWatch alarm.	`cloudwatchAlarm`
CloudWatch Logs	Sends a message to Amazon CloudWatch Logs.	`cloudwatchLogs`
CloudWatch metrics	Sends a message to a CloudWatch metric.	`cloudwatchMetric`
DynamoDB	Sends a message to a DynamoDB table.	`dynamoDB`
DynamoDBv2	Sends message data to multiple columns in a DynamoDB table.	`dynamoDBv2`

Rule action	Description	Name in API
Elasticsearch	Sends a message to an OpenSearch endpoint.	`OpenSearch`
HTTP	Posts a message to an HTTPS endpoint.	`http`
AWS IoT Events	Sends a message to an AWS IoT Events input.	`iotEvents`
AWS IoT SiteWise	Sends message data to AWS IoT SiteWise asset properties.	`iotSiteWise`
Firehose	Sends a message to a Firehose delivery stream.	`firehose`
Kinesis Data Streams	Sends a message to a Kinesis data stream.	`kinesis`
Lambda	Invokes a Lambda function with message data as input.	`lambda`
Location	Sends location data to Amazon Location Service.	`location`
OpenSearch	Sends a message to an Amazon OpenSearch Service endpoint.	`OpenSearch`
Republish	Republishes a message to another MQTT topic.	`republish`
S3	Stores a message in an Amazon Simple Storage Service (Amazon S3) bucket.	`s3`
Salesforce IoT	Sends a message to a Salesforce IoT input stream.	`salesforce`

Rule action	Description	Name in API
SNS	Publishes a message as an Amazon Simple Notification Service (Amazon SNS) push notification.	`sns`
SQS	Sends a message to an Amazon Simple Queue Service (Amazon SQS) queue.	`sqs`
Step Functions	Starts an AWS Step Functions state machine.	`stepFunctions`
the section called “Timestream”	Sends a message to an Amazon Timestream database table.	`timestream`

Notes

- Define the rule in the same AWS Region as another service's resource so that the rule action can interact with that resource.
- The AWS IoT rules engine might make multiple attempts to perform an action if intermittent errors occur. If all attempts fail, the message is discarded and the error is available in your CloudWatch Logs. You can specify an error action for each rule that is invoked after a failure occurs. For more information, see Error handling (error action).
- Some rule actions activate actions in services that integrate with AWS Key Management Service (AWS KMS) to support data encryption at rest. If you use a customer-managed AWS KMS key (KMS key) to encrypt data at rest, the service must have permission to use the KMS key on the caller's behalf. To learn how to manage permissions for your customer managed KMS key, see the data encryption topics in the appropriate service guide. For more information about customer managed KMS keys, see AWS Key Management Service concepts in the *AWS Key Management Service Developer Guide*.

  You can use any function or substitution template in an error action's SQL statement including the external functions: `aws_lambda()`, `get_dynamodb()`, get_registry_data(), get_thing_shadow(), get_secret(), machinelearning_predict(), and decode(). If an error action requires to call an external function, then invoking the error action can result in additional bill for the external function.

Apache Kafka

The Apache Kafka (Kafka) action sends messages directly to your Amazon Managed Streaming for Apache Kafka (Amazon MSK), Apache Kafka clusters managed by third-party providers such as Confluent Cloud, or self-managed Apache Kafka clusters. With Kafka rule action, you can route your IoT data to Kafka clusters. This enables you to build high-performance data pipelines for various purposes, such as streaming analytics, data integration, visualization, and mission-critical business applications.

Note

This topic assumes familiarity with the Apache Kafka platform and related concepts. For more information about Apache Kafka, see Apache Kafka. MSK Serverless is not supported. MSK Serverless clusters can only be done via IAM authentication, which Apache Kafka rule action doesn't currently support. For more information about how to configure AWS IoT Core with Confluent, see Leveraging Confluent and AWS to Solve IoT Device and Data Management Challenges.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `ec2:CreateNetworkInterface`, `ec2:DescribeNetworkInterfaces`, `ec2:CreateNetworkInterfacePermission`, `ec2:DeleteNetworkInterface`, `ec2:DescribeSubnets`, `ec2:DescribeVpcs`, `ec2:DescribeVpcAttribute`, and `ec2:DescribeSecurityGroups` operations. This role creates and manages elastic network interfaces to your Amazon Virtual Private Cloud to reach your Kafka broker. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT Core to perform this rule action.

For more information about network interfaces, see [Elastic network interfaces](#) in the *Amazon EC2 User Guide*.

The policy attached to the role that you specify should look like the following example.

```
{
    "Version":"2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "ec2:CreateNetworkInterface",
            "ec2:DescribeNetworkInterfaces",
            "ec2:CreateNetworkInterfacePermission",
            "ec2:DeleteNetworkInterface",
            "ec2:DescribeSubnets",
            "ec2:DescribeVpcs",
            "ec2:DescribeVpcAttribute",
            "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

- If you use AWS Secrets Manager to store the credentials required to connect to your Kafka broker, you must create an IAM role that AWS IoT Core can assume to perform the `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` operations.

  The policy attached to the role that you specify should look like the following example.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret"
            ],
            "Resource": [
```

```
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:kafka_client_truststore-*",
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:kafka_keytab-*"
            ]
        }
    ]
}
```

- You can run your Apache Kafka clusters inside Amazon Virtual Private Cloud (Amazon VPC). You must create an Apache Kafka Virtual Private Cloud (VPC) destination and use an NAT gateway in your subnets to forward messages from AWS IoT to a public Kafka cluster. The AWS IoT rules engine creates a network interface in each of the subnets listed in the destination to route traffic directly to the VPC. When you destination, the AWS IoT rules engine automatically creates a VPC rule action. For more information about VPC rule actions, see [Apache Kafka Virtual Private Cloud (VPC) destinations](#).
- If you use a customer managed AWS KMS key (KMS key) to encrypt data at rest, the service must have permission to use the KMS key on the caller's behalf. For more information, see [Amazon MSK encryption](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

destinationArn

The Amazon Resource Name (ARN) of the Apache Kafka Virtual Private Cloud (VPC) destination. For information about creating a destination, see [Apache Kafka Virtual Private Cloud (VPC) destinations](#).

topic

The Kafka topic for messages to be sent to the Kafka broker.

You can substitute this field using a substitution template. For more information, see [the section called “Substitution templates”](#).

key (optional)

The Kafka message key.

You can substitute this field using a substitution template. For more information, see the section called “Substitution templates”.

headers (optional)

The list of Kafka headers that you specify. Each header is a key-value pair that you can specify when you create a Kafka action. You can use these headers to route data from IoT clients to downstream Kafka clusters without modifying your message payload.

You can substitute this field using a substitution template. To understand how to pass an inline Rule's function as a substitution template in Kafka Action's header, see Examples. For more information, see the section called “Substitution templates”.

> **Note**
>
> Headers in binary format are not supported.

partition (optional)

The Kafka message partition.

You can substitute this field using a substitution template. For more information, see the section called “Substitution templates”.

clientProperties

An object that defines the properties of the Apache Kafka producer client.

acks (optional)

The number of acknowledgments the producer requires the server to have received before considering a request complete.

If you specify 0 as the value, the producer won't wait for any acknowledgment from the server. If the server doesn't receive the message, the producer won't retry to send the message.

Valid values: `-1`, `0`, `1`, `all`. The default value is `1`.

bootstrap.servers

A list of host and port pairs (for example, `host1:port1`, `host2:port2`) used to establish the initial connection to your Kafka cluster.

compression.type (optional)

The compression type for all data generated by the producer.

Valid values: `none`, `gzip`, `snappy`, `lz4`, `zstd`. The default value is `none`.

security.protocol

The security protocol used to attach to your Kafka broker.

Valid values: `SSL`, `SASL_SSL`. The default value is `SSL`.

key.serializer

Specifies how to turn the key objects that you provide with the`ProducerRecord` into bytes.

Valid value: `StringSerializer`.

value.serializer

Specifies how to turn value objects that you provide with the `ProducerRecord` into bytes.

Valid value: `ByteBufferSerializer`.

ssl.truststore

The truststore file in base64 format or the location of the truststore file in AWS Secrets Manager. This value isn't required if your truststore is trusted by Amazon certificate authorities (CA).

This field supports substitution templates. If you use Secrets Manager to store the credentials required to connect to your Kafka broker, you can use the `get_secret` SQL function to retrieve the value for this field. For more information about substitution templates, see the section called "Substitution templates". For more information about the `get_secret` SQL function, see the section called "get_secret(secretId, secretType, key, roleArn)". If the truststore is in the form of a file, use the `SecretBinary` parameter. If the truststore is in the form of a string, use the `SecretString` parameter.

The maximum size of this value is 65 KB.

ssl.truststore.password

The password for the truststore. This value is required only if you've created a password for the truststore.

ssl.keystore

The keystore file. This value is required when you specify `SSL` as the value for `security.protocol`.

This field supports substitution templates. Use Secrets Manager to store the credentials required to connect to your Kafka broker. To retrieve the value for this field, use the `get_secret` SQL function. For more information about substitution templates, see the section called "Substitution templates". For more information about the `get_secret` SQL function, see the section called "get_secret(secretId, secretType, key, roleArn)". Use the `SecretBinary` parameter.

ssl.keystore.password

The store password for the keystore file. This value is required if you specify a value for `ssl.keystore`.

The value of this field can be plaintext . This field also supports substitution templates. Use Secrets Manager to store the credentials required to connect to your Kafka broker. To retrieve the value for this field, use the `get_secret` SQL function. For more information about substitution templates, see the section called "Substitution templates". For more information about the `get_secret` SQL function, see the section called "get_secret(secretId, secretType, key, roleArn)". Use the `SecretString` parameter.

ssl.key.password

The password of the private key in your keystore file.

This field supports substitution templates. Use Secrets Manager to store the credentials required to connect to your Kafka broker. To retrieve the value for this field, use the `get_secret` SQL function. For more information about substitution templates, see the section called "Substitution templates". For more information about the `get_secret` SQL function, see the section called "get_secret(secretId, secretType, key, roleArn)". Use the `SecretString` parameter.

sasl.mechanism

The security mechanism used to connect to your Kafka broker. This value is required when you specify `SASL_SSL` for `security.protocol`.

Valid values: `PLAIN`, `SCRAM-SHA-512`, `GSSAPI`.

> **Note**
>
> `SCRAM-SHA-512` is the only supported security mechanism in the cn-north-1, cn-northwest-1, us-gov-east-1, and us-gov-west-1 Regions.

sasl.plain.username

The username used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `PLAIN` for `sasl.mechanism`.

sasl.plain.password

The password used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `PLAIN` for `sasl.mechanism`.

sasl.scram.username

The username used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `SCRAM-SHA-512` for `sasl.mechanism`.

sasl.scram.password

The password used to retrieve the secret string from Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `SCRAM-SHA-512` for `sasl.mechanism`.

sasl.kerberos.keytab

The keytab file for Kerberos authentication in Secrets Manager. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

This field supports substitution templates. Use Secrets Manager to store the credentials required to connect to your Kafka broker. To retrieve the value for this field, use the `get_secret` SQL function. For more information about substitution templates, see the section called "Substitution templates". For more information about the `get_secret` SQL function, see the section called "get_secret(secretId, secretType, key, roleArn)". Use the `SecretBinary` parameter.

sasl.kerberos.service.name

The Kerberos principal name under which Apache Kafka runs. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.krb5.kdc

The hostname of the key distribution center (KDC) to which your Apache Kafka producer client connects. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.krb5.realm

The realm to which your Apache Kafka producer client connects. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

sasl.kerberos.principal

The unique Kerberos identity to which Kerberos can assign tickets to access Kerberos-aware services. This value is required when you specify `SASL_SSL` for `security.protocol` and `GSSAPI` for `sasl.mechanism`.

Examples

The following JSON example defines an Apache Kafka action in an AWS IoT rule. The following example passes the sourceIp() inline function as a substitution template in the Kafka Action header.

```
{
 "topicRulePayload": {
  "sql": "SELECT * FROM 'some/topic'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
   {
    "kafka": {
     "destinationArn": "arn:aws:iot:region:123456789012:ruledestination/vpc/
VPCDestinationARN",
     "topic": "TopicName",
     "clientProperties": {
      "bootstrap.servers": "kafka.com:9092",
      "security.protocol": "SASL_SSL",
      "ssl.truststore": "${get_secret('kafka_client_truststore',
 'SecretBinary','arn:aws:iam::123456789012:role/kafka-get-secret-role-name')}",
      "ssl.truststore.password": "kafka password",
      "sasl.mechanism": "GSSAPI",
      "sasl.kerberos.service.name": "kafka",
```

```
      "sasl.kerberos.krb5.kdc": "kerberosdns.com",
      "sasl.kerberos.keytab": "${get_secret('kafka_keytab','SecretBinary',
 'arn:aws:iam::123456789012:role/kafka-get-secret-role-name')}",
      "sasl.kerberos.krb5.realm": "KERBEROSREALM",
      "sasl.kerberos.principal": "kafka-keytab/kafka-keytab.com"
     },
     "headers": [
      {
       "key": "static_header_key",
       "value": "static_header_value"
      },
      {
       "key": "substitutable_header_key",
       "value": "${value_from_payload}"
      },
      {
       "key": "source_ip",
       "value": "${sourceIp()}"
      }
     ]
    }
   }
  ]
 }
}
```

Important notes about your Kerberos setup

- Your key distribution center (KDC) must be resolvable through private Domain Name System (DNS) within your target VPC. One possible approach is to add the KDC DNS entry to a private hosted zone. For more information about this approach, see Working with private hosted zones.
- Each VPC must have DNS resolution enabled. For more information, see Using DNS with your VPC.
- Network interface security groups and instance-level security groups in the VPC destination must allow traffic from within your VPC on the following ports.
  - TCP traffic on the bootstrap broker listener port (often 9092, but must be within the 9000–9100 range)
  - TCP and UDP traffic on port 88 for the KDC
- `SCRAM-SHA-512` is the only supported security mechanism in the cn-north-1, cn-northwest-1, us-gov-east-1, and us-gov-west-1 Regions.

Apache Kafka Virtual Private Cloud (VPC) destinations

The Apache Kafka rule action routes data to an Apache Kafka cluster in an Amazon Virtual Private Cloud (Amazon VPC). The VPC configuration used by the Apache Kafka rule action is automatically enabled when you specify the VPC destination for your rule action.

An Apache Kafka Virtual Private Cloud (VPC) destination contains a list of subnets inside the VPC. The rules engine creates an elastic network interface in each subnet that you specify in this list. For more information about network interfaces, see Elastic network interfaces in the Amazon EC2 User Guide.

Requirements and considerations

- If you're using a self-managed Apache Kafka cluster that will be accessed using a public endpoint across the internet:
  - Create a NAT gateway for instances in your subnets. The NAT gateway has a public IP address that can connect to the internet, which allows the rules engine to forward your messages to the public Kafka cluster.
  - Allocate an Elastic IP address with the elastic network interfaces (ENIs) that are created by the Apache Kafka Virtual Private Cloud (VPC) destination. The security groups that you use must be configured to block incoming traffic.

  > **Note**
  >
  > If the Apache Kafka Virtual Private Cloud (VPC) destination is disabled and then re-enabled, you must re-associate the elastic IPs with the new ENIs.

- If an Apache Kafka Virtual Private Cloud (VPC) destination doesn't receive any traffic for 30 days in a row, it will be disabled.
- If any resources used by the Apache Kafka Virtual Private Cloud (VPC) destination change, the destination will be disabled and unable to be used.
- Some changes that can disable a Apache Kafka Virtual Private Cloud (VPC) destination include:
  - Deleting the VPC, subnets, security groups, or the role used.
  - Modifying the role to no longer have the necessary permissions.
  - Reaching near subnet capacity which makes us unable to apply FedRAMP patching.
  - Disabling the destination.

Pricing

For pricing purposes, a VPC rule action is metered in addition to the action that sends a message to a resource when the resource is in your VPC. For pricing information, see AWS IoT Core pricing.

Creating Apache Kafka Virtual Private Cloud (VPC) destinations

You create a Apache Kafka Virtual Private Cloud (VPC) destination by using the CreateTopicRuleDestination API or the AWS IoT Core console.

When you create a destination, you must specify the following information.

vpcId

The unique ID of the Amazon VPC.

subnetIds

A list of subnets in which the rules engine creates elastic network interfaces. The rules engine allocates a single network interface for each subnet in the list.

securityGroups (optional)

A list of security groups to apply to the network interfaces.

roleArn

The Amazon Resource Name (ARN) of a role that has permission to create network interfaces on your behalf.

This ARN should have a policy attached to it that looks like the following example.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
```

```
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterfacePermission",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "ec2:ResourceTag/VPCDestinationENI": "true"
            }
        }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateTags"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "ec2:CreateAction": "CreateNetworkInterface",
                    "aws:RequestTag/VPCDestinationENI": "true"
            }
        }
        }
    ]
}
```

Creating an Apache Kafka Virtual Private Cloud (VPC) destination by using AWS CLI

The following example shows how to create a destination by using AWS CLI.

```
aws --region regions iot create-topic-rule-destination --destination-configuration
 'vpcConfiguration={subnetIds=["subnet-
123456789101230456"],securityGroups=[],vpcId="vpc-
123456789101230456",roleArn="arn:aws:iam::123456789012:role/role-name"}'
```

After you run this command, the destination status will be IN_PROGRESS. After a few minutes, its status will change to either ERROR (if the command isn't successful) or ENABLED. When the destination status is ENABLED, it's ready to use.

You can use the following command to get the status of your Apache Kafka Virtual Private Cloud (VPC) destination.

```
aws --region region iot get-topic-rule-destination --arn "VPCDestinationARN"
```

Creating a Apache Kafka Virtual Private Cloud (VPC) destination by using the AWS IoT Core console

The following steps describe how to create a destination by using the AWS IoT Core console.

1. Navigate to the AWS IoT Core console. In the left pane, on the **Act** tab, choose **Destinations**.
2. Enter values for the following fields.
   - **VPC ID**
   - **Subnet IDs**
   - **Security Group**
3. Select a role that has the permissions required to create network interfaces. The preceding example policy contains these permissions.

When the Apache Kafka Virtual Private Cloud (VPC) destination status is **ENABLED**, it's ready to use.

CloudWatch alarms

The CloudWatch alarm (cloudWatchAlarm) action changes the state of an Amazon CloudWatch alarm. You can specify the state change reason and value in this call.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the cloudwatch:SetAlarmState operation. For more information, see Granting an AWS IoT rule the access it requires.

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`alarmName`

The CloudWatch alarm name.

Supports substitution templates: API and AWS CLI only

`stateReason`

Reason for the alarm change.

Supports substitution templates: Yes

`stateValue`

The value of the alarm state. Valid values: `OK`, `ALARM`, `INSUFFICIENT_DATA`.

Supports substitution templates: Yes

`roleArn`

The IAM role that allows access to the CloudWatch alarm. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a CloudWatch alarm action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
```

```
        "actions": [
            {
                "cloudwatchAlarm": {
                    "alarmName": "IotAlarm",
                    "stateReason": "Temperature stabilized.",
                    "stateValue": "OK",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"
                }
            }
        ]
    }
}
```

See also

- What is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*
- Using Amazon CloudWatch alarms in the *Amazon CloudWatch User Guide*

CloudWatch Logs

The CloudWatch Logs (`cloudwatchLogs`) action sends data to Amazon CloudWatch Logs. You can use `batchMode` to upload and timestamp multiple device log records in one message. You can also specify the log group where the action sends data.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `logs:CreateLogStream`, `logs:DescribeLogStreams`, and `logs:PutLogEvents` operations. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use a customer managed AWS KMS key (KMS key) to encrypt log data in CloudWatch Logs, the service must have permission to use the KMS key on the caller's behalf. For more information, see Encrypt log data in CloudWatch Logs using AWS KMS in the *Amazon CloudWatch Logs User Guide*.

MQTT message format requirements for `batchMode`

If you use the CloudWatch Logs rule action with `batchMode` turned off, there are no MQTT message formatting requirements. (Note: the `batchMode` parameter's default value is `false`.) However, if you use the CloudWatch Logs rule action with `batchMode` turned on (the parameter value is `true`), MQTT messages containing device-side logs must be formatted to contain a timestamp and a message payload. **Note:** `timestamp` represents the time that the event occurred and is expressed as a number of milliseconds after January 1, 1970 00:00:00 UTC.

The following is an example of the publish format:

```
[
  {"timestamp": 1673520691093, "message": "Test message 1"},
  {"timestamp": 1673520692879, "message": "Test message 2"},
  {"timestamp": 1673520693442, "message": "Test message 3"}
]
```

Depending on how the device-side logs are generated, they might need to be filtered and reformatted before they're sent to comply with this requirement. For more information, see MQTT Message payload.

Independent of the `batchMode` parameter, `message` contents must comply with AWS IoT message size limitations. For more information, see AWS IoT Core endpoints and quotas.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`logGroupName`

The CloudWatch log group where the action sends data.

Supports substitution templates: API and AWS CLI only

`roleArn`

The IAM role that allows access to the CloudWatch log group. For more information, see Requirements.

Supports substitution templates: No

(optional) `batchMode`

Indicates whether batches of log records will be extracted and uploaded into CloudWatch. Values include `true` or `false` (default). For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a CloudWatch Logs action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "cloudwatchLogs": {
                    "logGroupName": "IotLogs",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw",
                    "batchMode": false
                }
            }
        ]
    }
}
```

See also

- What is Amazon CloudWatch Logs? in the *Amazon CloudWatch Logs User Guide*

CloudWatch metrics

The CloudWatch metric (`cloudwatchMetric`) action captures an Amazon CloudWatch metric. You can specify the metric namespace, name, value, unit, and timestamp.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `cloudwatch:PutMetricData` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`metricName`

The CloudWatch metric name.

Supports substitution templates: Yes

`metricNamespace`

The CloudWatch metric namespace name.

Supports substitution templates: Yes

`metricUnit`

The metric unit supported by CloudWatch.

Supports substitution templates: Yes

`metricValue`

A string that contains the CloudWatch metric value.

Supports substitution templates: Yes

`metricTimestamp`

(Optional) A string that contains the timestamp, expressed in seconds in Unix epoch time. Defaults to the current Unix epoch time.

Supports substitution templates: Yes

`roleArn`

The IAM role that allows access to the CloudWatch metric. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a CloudWatch metric action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "cloudwatchMetric": {
                    "metricName": "IotMetric",
                    "metricNamespace": "IotNamespace",
                    "metricUnit": "Count",
                    "metricValue": "1",
                    "metricTimestamp": "1456821314",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"
                }
            }
        ]
    }
}
```

The following JSON example defines a CloudWatch metric action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "cloudwatchMetric": {
                    "metricName": "${topic()}",
                    "metricNamespace": "${namespace}",
                    "metricUnit": "${unit}",
                    "metricValue": "${value}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_cw"
                }
            }
        ]
```

```
    }
}
```

See also

- [What is Amazon CloudWatch?] in the *Amazon CloudWatch User Guide*
- [Using Amazon CloudWatch metrics] in the *Amazon CloudWatch User Guide*

DynamoDB

The DynamoDB (`dynamoDB`) action writes all or part of an MQTT message to an Amazon DynamoDB table.

You can follow a tutorial that shows you how to create and test a rule with a DynamoDB action. For more information, see Tutorial: Storing device data in a DynamoDB table.

> **Note**
>
> This rule writes non-JSON data to DynamoDB as binary data. The DynamoDB console displays the data as base64-encoded text.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `dynamodb:PutItem` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use a customer managed AWS KMS key (KMS key) to encrypt data at rest in DynamoDB, the service must have permission to use the KMS key on the caller's behalf. For more information, see Customer Managed KMS key in the *Amazon DynamoDB Getting Started Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`tableName`

The name of the DynamoDB table.

Supports substitution templates: API and AWS CLI only

`hashKeyField`

The name of the hash key (also called the partition key).

Supports substitution templates: API and AWS CLI only

`hashKeyType`

(Optional) The data type of the hash key (also called the partition key). Valid values: `STRING`, `NUMBER`.

Supports substitution templates: API and AWS CLI only

`hashKeyValue`

The value of the hash key. Consider using a substitution template such as `${topic()}` or `${timestamp()}`.

Supports substitution templates: Yes

`rangeKeyField`

(Optional) The name of the range key (also called the sort key).

Supports substitution templates: API and AWS CLI only

`rangeKeyType`

(Optional) The data type of the range key (also called the sort key). Valid values: `STRING`, `NUMBER`.

Supports substitution templates: API and AWS CLI only

`rangeKeyValue`

(Optional) The value of the range key. Consider using a substitution template such as `${topic()}` or `${timestamp()}`.

Supports substitution templates: Yes

`payloadField`

(Optional) The name of the column where the payload is written. If you omit this value, the payload is written to the column named `payload`.

Supports substitution templates: Yes

`operation`

(Optional) The type of operation to be performed. Valid values: `INSERT`, `UPDATE`, `DELETE`.

Supports substitution templates: Yes

`roleARN`

The IAM role that allows access to the DynamoDB table. For more information, see Requirements.

Supports substitution templates: No

The data written to the DynamoDB table is the result from the SQL statement of the rule.

Examples

The following JSON example defines a DynamoDB action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * AS message FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "dynamoDB": {
                    "tableName": "my_ddb_table",
                    "hashKeyField": "key",
                    "hashKeyValue": "${topic()}",
                    "rangeKeyField": "timestamp",
                    "rangeKeyValue": "${timestamp()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB"
                }
            }
        ]
```

```
    }
}
```

See also

- What is Amazon DynamoDB? in the *Amazon DynamoDB Developer Guide*
- Getting started with DynamoDB in the *Amazon DynamoDB Developer Guide*
- Tutorial: Storing device data in a DynamoDB table

DynamoDBv2

The DynamoDBv2 (`dynamoDBv2`) action writes all or part of an MQTT message to an Amazon DynamoDB table. Each attribute in the payload is written to a separate column in the DynamoDB database.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `dynamodb:PutItem` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- The MQTT message payload must contain a root-level key that matches the table's primary partition key and a root-level key that matches the table's primary sort key, if one is defined.
- If you use a customer managed AWS KMS key (KMS key) to encrypt data at rest in DynamoDB, the service must have permission to use the KMS key on the caller's behalf. For more information, see Customer Managed KMS key in the *Amazon DynamoDB Getting Started Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`putItem`

An object that specifies the DynamoDB table to which the message data will be written. This object must contain the following information:

`tableName`

The name of the DynamoDB table.

Supports substitution templates: API and AWS CLI only

`roleARN`

The IAM role that allows access to the DynamoDB table. For more information, see Requirements.

Supports substitution templates: No

The data written to the DynamoDB table is the result from the SQL statement of the rule.

Examples

The following JSON example defines a DynamoDBv2 action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * AS message FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "dynamoDBv2": {
                    "putItem": {
                        "tableName": "my_ddb_table"
                    },
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDBv2",
                }
            }
        ]
    }
}
```

The following JSON example defines a DynamoDB action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
```

```
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2015-10-08",
        "actions": [
            {
                "dynamoDBv2": {
                    "putItem": {
                        "tableName": "${topic()}"
                    },
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDBv2"
                }
            }
        ]
    }
}
```

See also

- What is Amazon DynamoDB? in the *Amazon DynamoDB Developer Guide*
- Getting started with DynamoDB in the *Amazon DynamoDB Developer Guide*

Elasticsearch

The Elasticsearch (`elasticsearch`) action writes data from MQTT messages to an Amazon OpenSearch Service domain. You can then use tools like OpenSearch Dashboards to query and visualize data in OpenSearch Service.

> **Warning**
>
> The `Elasticsearch` action can only be used by existing rule actions. To create a new rule action or to update an existing rule action, use the `OpenSearch` rule action instead. For more information, see OpenSearch.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `es:ESHttpPut` operation. For more information, see Granting an AWS IoT rule the access it requires.

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use a customer managed AWS KMS key (KMS key) to encrypt data at rest in OpenSearch, the service must have permission to use the KMS key on the caller's behalf. For more information, see Encryption of data at rest for Amazon OpenSearch Service in the *Amazon OpenSearch Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`endpoint`

The endpoint of your service domain.

Supports substitution templates: API and AWS CLI only

`index`

The index where you want to store your data.

Supports substitution templates: Yes

`type`

The type of document you are storing.

Supports substitution templates: Yes

`id`

The unique identifier for each document.

Supports substitution templates: Yes

`roleARN`

The IAM role that allows access to the OpenSearch Service domain. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines an Elasticsearch action in an AWS IoT rule and how you can specify the fields for the `elasticsearch` action. For more information, see ElasticsearchAction.

```
{
    "topicRulePayload": {
        "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "elasticsearch": {
                    "endpoint": "https://my-endpoint",
                    "index": "my-index",
                    "type": "my-type",
                    "id": "${newuuid()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es"
                }
            }
        ]
    }
}
```

The following JSON example defines an Elasticsearch action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "elasticsearch": {
                    "endpoint": "https://my-endpoint",
                    "index": "${topic()}",
                    "type": "${type}",
                    "id": "${newuuid()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_es"
                }
            }
        ]
```

```
    }
}
```

See also

- OpenSearch
- What is Amazon OpenSearch Service?

HTTP

The HTTPS (`http`) action sends data from an MQTT message to an HTTPS endpoint, which can point to a web application or service.

Requirements

This rule action has the following requirements:

- You must confirm and enable HTTPS endpoints before the rules engine can use them. For more information, see HTTP action destinations.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`url`

The HTTPS endpoint where the message is sent using the HTTP POST method. If you use an IP address in place of a hostname, it must be an IPv4 address. IPv6 addresses are not supported.

Supports substitution templates: Yes

`confirmationUrl`

(Optional) If specified, AWS IoT uses the confirmation URL to create a matching topic rule destination. You must enable the HTTP action destination before using it in an HTTP action. For more information, see HTTP action destinations. If you use substitution templates, you must manually create an HTTP action destination before the `http` action can be used. `confirmationUrl` must be a prefix of `url`.

The relationship between `url` and `confirmationUrl` is described by the following:

- If `url` is hardcoded and `confirmationUrl` is not provided, we implicitly treat the `url` field as the `confirmationUrl`. AWS IoT creates a topic rule destination for `url`.
- If `url` and `confirmationUrl` are hardcoded, `url` must begin with `confirmationUrl`. AWS IoT creates a topic rule destination for `confirmationUrl`.
- If `url` contains a substitution template, you must specify `confirmationUrl` and `url` must begin with `confirmationUrl`. If `confirmationUrl` contains substitution templates, you must manually create an HTTP action destination before the `http` action can be used. If `confirmationUrl` does not contain substitution templates, AWS IoT creates a topic rule destination for `confirmationUrl`.

Supports substitution templates: Yes

`headers`

(Optional) The list of headers to include in HTTP requests to the endpoint. Each header must contain the following information:

`key`

The key of the header.

Supports substitution templates: No

`value`

The value of the header.

Supports substitution templates: Yes

> **Note**
>
> The default content type is application/json when the payload is in JSON format. Otherwise, it is application/octet-stream. You can overwrite it by specifying the exact content type in the header with the key content-type (case insensitive).

`auth`

(Optional) The authentication used by the rules engine to connect to the endpoint URL specified in the `url` argument. Currently, Signature Version 4 is the only supported authentication type. For more information, see HTTP Authorization.

Supports substitution templates: No

`enableBatching`

(Optional) Whether to process the HTTP action messages into a single request for a given url. Value can be true or false. For more information on batching, see Batching HTTP action messages.

Boolean value

Supports substitution templates: No

`batchConfig`

(Optional) Configuration settings for batching. Once enabled, `batchConfig` parameters must be specified. If `batchConfig` parameters are not specified, the default values will be used.

`maxBatchOpenMs`

The maximum amount of time (in milliseconds) an outgoing message waits for other messages to create the batch. The higher the setting, the longer the latency of the batched HTTP action.

Minimum Value: 5 ms. Maximum Value: 200 ms.

Default Value: 20 ms

Supports substitution templates: No

`maxBatchSize`

The maximum number of messages that are batched together in a single action execution.

Supports substitution templates: No

Minimum Value: 2 messages. Maximum Value: 10 messages

Default Value: 10 messages

`maxBatchSizeBytes`

Maximum size of a message batch, in bytes.

Minimum Value: 100 bytes. Maximum Value: 131,072 bytes

Default Value: 5,120 bytes

Supports substitution templates: No

Note

The default content type is application/json when the payload is in JSON format. Otherwise, it is application/octet-stream. You can overwrite it by specifying the exact content type in the header with the key content-type (case insensitive).

Examples

The following JSON example defines an AWS IoT rule with an HTTP action.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "http": {
                    "url": "https://www.example.com/subpath",
                    "confirmationUrl": "https://www.example.com",
                    "headers": [
                        {
                            "key": "static_header_key",
                            "value": "static_header_value"
                        },
                        {
                            "key": "substitutable_header_key",
                            "value": "${value_from_payload}"
                        }
                    ]
                }
            }
        ]
    }
}
```

```
"http": {
    "url": "https://www.example.com/subpath",
    "confirmationUrl": "https://www.example.com",
    "headers": [
        {
```

```
            "key": "Content-Type",
            "value": "application/json"
          }
    ],
    "enableBatching": true,
    "batchConfig": {
      "maxBatchOpenMs": 123,
      "maxBatchSize": 5,
      "maxBatchSizeBytes": 131072,
     }
 },
 "errorAction": {
        "http": {
            "url": "https://www.example.com/subpath",
            "confirmationUrl": "https://www.example.com"
            // batchConfig is not allowed here
        }
}
```

HTTP action retry logic

The AWS IoT rules engine retries the HTTP action according to these rules:

- The rules engine tries to send a message at least once.
- The rules engine retries at most twice. The maximum number of tries is three.
- The rules engine does not attempt a retry if:
  - The previous try provided a response larger than 16,384 bytes.
  - The downstream web service or application closes the TCP connection after the try.
  - The total time to complete a request with retries exceeded the request timeout limit.
  - The request returns an HTTP status code other than 429, 500-599.

> **Note**
>
> Standard data transfer costs apply to retries.

See also

- Batching HTTP action messages

- HTTP action destinations
- Route data directly from AWS IoT Core to your web services in the *Internet of Things on AWS* blog

Batching HTTP action messages

You can use batching to send multiple HTTP action messages in a single request.

Overview

Batching enables you to send messages from AWS IoT Core Rules Engine to your HTTP endpoints in batches. This functionality can help reduce your costs by lowering the number of HTTP action executions as well as improve efficiency by reducing the overhead associated with establishing new connections.

> **Note**
>
> The batched HTTP action is metered as a single action. You are metered in increments of 5 kiB, based on the size of outbound batched payload emitted by the AWS IoT Core Rules Engine to the downstream service. For more information, see the AWS IoT Core pricing page.

When you enable batching in the definition of your IoT Rule Action, the following parameters will be available for configuration:

`maxBatchOpenMs`

The maximum amount of time (in milliseconds) an outgoing message waits for other messages to create the batch. The higher the setting, the longer the latency of the batched HTTP action.

Minimum Value: 5 ms. Maximum Value: 200 ms.

Default Value: 20 ms

Supports substitution templates: No

`maxBatchSize`

The maximum number of messages that are batched together in a single IoT rule action execution.

Minimum Value: 2 messages. Maximum Value: 10 messages

Default Value: 10 messages

Supports substitution templates: No

`maxBatchSizeBytes`

Maximum size of a message batch, in bytes.

Minimum Value: 100 bytes. Maximum Value: 131,072 bytes

Default Value: 5120 bytes

Supports substitution templates: No

> **Important**
>
> When you specify multiple batch parameters, batching completes when the first limit is reached. For example, if you specify 100 ms as the Maximum Batch Open Time and 5 kiB as the Maximum Batch Size, and Rules Engine batches only 2 kiB within 100 ms, then a 2 kiB batch will be created and sent.

Using HTTP headers in a batch

When you use headers in your HTTP action, the batched request uses the header value from the last message that was added to the batch (not necessarily the last message you published). We recommend using header values that are either:

- Identical across all messages in the batch
- Applicable to all messages (for example, authentication credentials)

The headers are sent with the HTTP request and are not part of the message body.

> **Note**
>
> When batching is enabled:
>
> - The batched request automatically includes the `Content-Type: application/json` header, as the batch is sent as a JSON array.

- We can't guarantee that the last message in the batch is the last message that you published. It is the last message that made it into the batch.

Payload Example

The following example shows the structure of a batched message payload sent to your HTTP endpoint:

```
[
  {
    "user_id": "user1",
    "steps_today": 1000
  },
  {
    "user_id": "user2",
    "steps_today": 21000
  },
  {
    "user_id": "user8",
    "steps_today": 1500
  },
  ...
]
```

Limitations

The following are limitations on batching:

- AWS IoT Core does not guarantee overall message ordering. Batching is performed locally on each host, which may result in messages within a batch being processed in a different order than they were received.
- AWS IoT Core does not provide message processing support on the receiver side. You are responsible for ensuring that your downstream service is configured to accept and process data in batches.
- Cross-account batching is not supported, even if messages are destined for the same resource identifier (HTTP URL or resource ARN).
- AWS IoT Core does not guarantee that the batch size will meet the configuration you specified. Batches may be smaller than your configured limits based on timing and message flow.

- When batching is enabled, binary payloads (non-UTF-8 data) are not supported. Only UTF-8 text payloads (such as JSON) are accepted. To send binary data, base64 encode it before sending it to the HTTP action, and then decode it at your receiving endpoint. For example, you can use the encode function in IoT rules to encode the binary payload. Alternatively, you can encode the binary payload in your IoT device and publish it to AWS IoT Core.

Error Actions for Batching

You will not be able to define a separate batching logic in your Error Action definition. However, your Error Action will support batching if you have defined batching logic in your primary Action.

When a batch request fails, AWS IoT Core Rules engine will follow the HTTP action retry logic. After the final retry attempt, an error action will be invoked for the entire failed batch.

The following is an example of an error action message with batching enabled:

```
{
    "ruleName": "FailedTopicRule",
    "topic": "topic/rulesengine",
    "payloadsWithMetadata": [
        {
            "id": 1,
            "cloudwatchTraceId": "bebd6d93-6d4a-899e-9e40-56e82252d2be",
            "clientId": "Test",
            "sourceIp": "10.0.0.0",
            "base64OriginalPayload":
 "eyJ1c2VyX2lkIjogInVzZXI1NjQ3IiwgInN0ZXBzX3RvZGF5IjogMTMzNjUsICJ0aW1lc3RhbXAiOiAiMjAyNS0xMC0wO
        },
        {
            "id": 2,
            "cloudwatchTraceId": "af94d3b8-0b18-1dbf-2c7d-513f5cb9e2e1",
            "clientId": "Test",
            "sourceIp": "10.0.0.0",
            "base64OriginalPayload":
 "eyJ1c2VyX2lkIjogInVzZXI1NjQ3IiwgInN0ZXBzX3RvZGF5IjogMTMzNjUsICJ0aW1lc3RhbXAiOiAiMjAyNS0xMC0wO
        },
        {
            "id": 3,
            "cloudwatchTraceId": "ca441266-c2ce-c916-6aee-b9e5c7831675",
            "clientId": "Test",
            "sourceIp": "10.0.0.0",
```

```
            "base64OriginalPayload":
 "eyJ1c2VyX2lkIjogInVzZXI1NjQ3IiwgInN0ZXBzX3RvZGF5IjogMTMzNjUsICJ0aW1lc3RhbXAiOiAiMjAyNS0xMC0w
        }
    ],
    "failures": [
        {
            "affectedIds": [
                1,
                2,
                3
            ],
            "failedAction": "HttpAction",
            "failedResource": "https://example.foobar.com/HttpAction",
            "errorMessage": "HttpAction failed to make a request to the specified
 endpoint. StatusCode: 500. Reason: Internal Server Error."
        },
        {
            "affectedIds": [
                3
            ],
            "failedAction": "S3Action",
            "failedResource": "amzn-s3-demo-bucket",
            "errorMessage": "Failed to put S3 object. The error received was The
 specified bucket does not exist"
        },
        {
            "affectedIds": [
                3
            ],
            "failedAction": "LambdaAction",
            "failedResource": "arn:aws:lambda:us-west-2:123456789012:function:dummy",
            "errorMessage": "Failed to invoke lambda function. Received Server error
 from Lambda. The error code is 403"
        }
    ]
}
```

Note

Batched action failures also generate larger error action payloads which can increase the probability of error action failures due to size. You can monitor error action failures using the `ErrorActionFailure` metric. See Rule action metrics for more information.

Batching HTTP action messages with the AWS CLI

Creating or updating a rule action with batching

1. Use the appropriate AWS CLI command to create or update a rule:
   - To create a new rule, use the create-topic-rule command:

```
aws iot create-topic-rule --rule-name myrule --topic-rule-payload
 file://myrule.json
```

   - To update an existing rule, use the replace-topic-rule command:

```
aws iot replace-topic-rule --rule-name myrule --topic-rule-payload
 file://myrule.json
```

2. Enable batching capabilities by setting the enableBatching parameter to true in your topic rule payload:

```
{
        "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "http": {
                    "url": "https://www.example.com/subpath",
                    "confirmationUrl": "https://www.example.com",
                    "headers": [
                        {
                            "key": "static_header_key",
                            "value": "static_header_value"
                        },
                        {
                            "key": "substitutable_header_key",
                            "value": "${value_from_payload}"
                         }
                    ],
                    "enableBatching": true,
                    "batchConfig": {
                       "maxBatchOpenMs": 100,
                       "maxBatchSize": 5,
                       "maxBatchSizeBytes": 1024
```

```
                    }
                }
            }
        ]
}
```

3. Configure the batching parameters. You do not need to specify all batch parameters. You can choose to specify 1, 2, or all 3 batch parameters. If you do not specify a batch parameter, Rules Engine will update that parameter with the default values. For more information on batching parameters and their default values, see HTTP parameters.

HTTP action destinations

An HTTP action destination is a web service to which the rules engine can route data from a topic rule. An AWS IoT Core resource describes the web service for AWS IoT. Destination resources can be shared by different rules.

Before AWS IoT Core can send data to another web service, it must confirm that it can access the service's endpoint.

Overview

An HTTP action destination refers to a web service that supports a confirmation URL and one or more data collection URLs. The destination resource contains the confirmation URL of your web service. When you configure an HTTP action, you specify the actual URL of the endpoint that should receive the data along with the web service's confirmation URL. After your destination is confirmed, the topic rule sends the result of the SQL statement to the HTTPS endpoint (and not to the confirmation URL).

An HTTP action destination can be in one of the following states:

ENABLED

The destination has been confirmed and can be used by a rule action. A destination must be in the `ENABLED` state for it to be used in a rule. You can only enable a destination that's in DISABLED status.

DISABLED

The destination has been confirmed but it can't be used by a rule action. This is useful if you want to temporarily prevent traffic to your endpoint without having to go through the confirmation process again. You can only disable a destination that's in ENABLED status.

IN_PROGRESS

Confirmation of the destination is in progress.

ERROR

Destination confirmation timed out.

After an HTTP action destination has been confirmed and enabled, it can be used with any rule in your account.

Managing HTTP action destinations

You can use the following operations to manage your HTTP action destinations.

Creating HTTP action destinations

You create an HTTP action destination by calling the `CreateTopicRuleDestination` operation or by using the AWS IoT console.

After you create a destination, AWS IoT sends a confirmation request to the confirmation URL. The confirmation request has the following format:

```
HTTP POST {confirmationUrl}/?confirmationToken={confirmationToken}
Headers:
x-amz-rules-engine-message-type: DestinationConfirmation
x-amz-rules-engine-destination-arn:"arn:aws:iot:us-east-1:123456789012:ruledestination/
http/7a280e37-b9c6-47a2-a751-0703693f46e4"
Content-Type: application/json
Body:
{
    "arn":"arn:aws:iot:us-east-1:123456789012:ruledestination/http/7a280e37-b9c6-47a2-
a751-0703693f46e4",
    "confirmationToken": "AYADeMXLrPrNY2wqJAKsFNn-…NBJndA",
    "enableUrl": "https://iot.us-east-1.amazonaws.com/confirmdestination/
AYADeMXLrPrNY2wqJAKsFNn-…NBJndA",
    "messageType": "DestinationConfirmation"
}
```

The content of the confirmation request includes the following information:

arn

The Amazon Resource Name (ARN) for the HTTP action destination to confirm.

confirmationToken

The confirmation token sent by AWS IoT Core. The token in the example is truncated. Your token will be longer. You'll need this token to confirm your destination with AWS IoT Core.

enableUrl

The URL to which you browse to confirm a topic rule destination.

messageType

The type of message.

Confirming HTTP action destinations

To complete the endpoint confirmation process, if you're using the AWS CLI, you must perform the following steps after your confirmation URL receives the confirmation request.

1. **Confirm that the destination is ready to receive messages**

   To confirm that the HTTP action destination is ready to receive IoT messages, either call the `enableUrl` in the confirmation request, or perform the `ConfirmTopicRuleDestination` API operation and pass the `confirmationToken` from the confirmation request.

2. **Set topic rule status to enabled**

   After you've confirmed that the destination can receive messages, you must perform the `UpdateTopicRuleDestination` API operation to set the status of the topic rule to `ENABLED`.

If you're using the AWS IoT console, copy the `confirmationToken` and paste it into the destination's confirmation dialog in the AWS IoT console. You can then enable the topic rule.

Sending a new confirmation request

To activate a new confirmation message for a destination, call `UpdateTopicRuleDestination` and set the topic rule destination's status to `IN_PROGRESS`.

Repeat the confirmation process after you send a new confirmation request.

Disabling and deleting an HTTP action destination

To disable a destination, call `UpdateTopicRuleDestination` and set the topic rule destination's status to `DISABLED`. A topic rule in the DISABLED state can be enabled again without the need to send a new confirmation request.

To delete an HTTP action destination, call `DeleteTopicRuleDestination`.

Certificate Authority Support

> **Note**
>
> Self-signed certificates are not supported.

HTTPS Endpoints in an HTTP action destination support certificates issued by both AWS Private Certificate Authority and Lets Encrypt.

AWS IoT Events

The AWS IoT Events (`iotEvents`) action sends data from an MQTT message to an AWS IoT Events input.

> **Important**
>
> If the payload is sent to AWS IoT Core without the `Input attribute Key`, or if the key isn't in the same JSON path specified in the key, it will cause the IoT rule to fail with the error `Failed to send message to Iot Events`.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iotevents:BatchPutMessage` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`batchMode`

(Optional) Whether to process the event actions as a batch. The default value is `false`.

When `batchMode` is `true` and the rule SQL statement evaluates to an Array, each Array element is treated as a separate message when it's sent to AWS IoT Events by calling `BatchPutMessage`. The resulting array can't have more than 10 messages.

When `batchMode` is `true`, you can't specify a `messageId`.

Supports substitution templates: No

`inputName`

The name of the AWS IoT Events input.

Supports substitution templates: API and AWS CLI only

`messageId`

(Optional) Use this to verify that only one input (message) with a given `messageId` is processed by an AWS IoT Events detector. You can use the `${newuuid()}` substitution template to generate a unique ID for each request.

When `batchMode` is `true`, you can't specify a `messageId`--a new UUID value will be assigned.

Supports substitution templates: Yes

`roleArn`

The IAM role that allows AWS IoT to send an input to an AWS IoT Events detector. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines an IoT Events action in an AWS IoT rule.

```
{
```

```
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "iotEvents": {
                    "inputName": "MyIoTEventsInput",
                    "messageId": "${newuuid()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_events"
                }
            }
        ]
    }
}
```

See also

- What is AWS IoT Events? in the *AWS IoT Events Developer Guide*

AWS IoT SiteWise

The AWS IoT SiteWise (`iotSiteWise`) action sends data from an MQTT message to asset properties in AWS IoT SiteWise.

You can follow a tutorial that shows you how to ingest data from AWS IoT things. For more information, see the Ingesting data to AWS IoT SiteWise from AWS IoT things tutorial or the Ingesting data using AWS IoT Core rules section in the *AWS IoT SiteWise User Guide*.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iotsitewise:BatchPutAssetPropertyValue` operation. For more information, see Granting an AWS IoT rule the access it requires.

  You can attach the following example trust policy to the role.

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*"
        }
    ]
}
```

To improve security, you can specify an AWS IoT SiteWise asset hierarchy path in the `Condition` property. The following example is a trust policy that specifies an asset hierarchy path.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "iotsitewise:assetHierarchyPath": [
                        "/root node asset ID",
                        "/root node asset ID/*"
                    ]
                }
            }
        }
    ]
}
```

- When you send data to AWS IoT SiteWise with this action, your data must meet the requirements of the `BatchPutAssetPropertyValue` operation. For more information, see BatchPutAssetPropertyValue in the *AWS IoT SiteWise API Reference*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`putAssetPropertyValueEntries`

A list of asset property value entries that each contain the following information:

`propertyAlias`

(Optional) The property alias associated with your asset property. Specify either a `propertyAlias` or both an `assetId` and a `propertyId`. For more information about property aliases, see [Mapping industrial data streams to asset properties] in the *AWS IoT SiteWise User Guide*.

Supports [substitution templates]: Yes

`assetId`

(Optional) The ID of the AWS IoT SiteWise asset. Specify either a `propertyAlias` or both an `assetId` and a `propertyId`.

Supports [substitution templates]: Yes

`propertyId`

(Optional) The ID of the asset's property. Specify either a `propertyAlias` or both an `assetId` and a `propertyId`.

Supports [substitution templates]: Yes

`entryId`

(Optional) A unique identifier for this entry. Define the `entryId` to better track which message caused an error if failure occurs. Defaults to a new UUID.

Supports [substitution templates]: Yes

`propertyValues`

A list of property values to insert that each contain timestamp, quality, and value (TQV) in the following format:

`timestamp`

A timestamp structure that contains the following information:

`timeInSeconds`

A string that contains the time in seconds in Unix epoch time. If your message payload doesn't have a timestamp, you can use [timestamp()], which returns the current time in milliseconds. To convert that time to seconds, you can use the following substitution template: **`${floor(timestamp() / 1E3)}`**.

Supports substitution templates: Yes

`offsetInNanos`

(Optional) A string that contains the nanosecond time offset from the time in seconds. If your message payload doesn't have a timestamp, you can use timestamp(), which returns the current time in milliseconds. To calculate the nanosecond offset from that time, you can use the following substitution template: **`${(timestamp() % 1E3) * 1E6}`**.

Supports substitution templates: Yes

Regarding Unix epoch time, AWS IoT SiteWise accepts only entries that have a timestamp of up to 7 days in the past up to 5 minutes in the future.

`quality`

(Optional) A string that describes the quality of the value. Valid values: `GOOD`, `BAD`, `UNCERTAIN`.

Supports substitution templates: Yes

`value`

A value structure that contains one of the following value fields, depending on the asset property's data type:

`booleanValue`

(Optional) A string that contains the Boolean value of the value entry.

Supports substitution templates: Yes

`doubleValue`

(Optional) A string that contains the double value of the value entry.

Supports substitution templates: Yes

`integerValue`

(Optional) A string that contains the integer value of the value entry.

Supports substitution templates: Yes

`stringValue`

(Optional) The string value of the value entry.

Supports substitution templates: Yes

`roleArn`

The ARN of the IAM role that grants AWS IoT permission to send an asset property value to AWS IoT SiteWise. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a basic IoT SiteWise action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "iotSiteWise": {
                    "putAssetPropertyValueEntries": [
                        {
                            "propertyAlias": "/some/property/alias",
                            "propertyValues": [
                                {
                                    "timestamp": {
                                        "timeInSeconds": "${my.payload.timeInSeconds}"
                                    },
                                    "value": {
                                        "integerValue": "${my.payload.value}"
                                    }
                                }
                            ]
                        }
                    ],
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sitewise"
                }
            }
        ]
```

```
    }
}
```

The following JSON example defines an IoT SiteWise action in an AWS IoT rule. This example uses the topic as the property alias and the `timestamp()` function. For example, if you publish data to `/company/windfarm/3/turbine/7/rpm`, this action sends the data to the asset property with a property alias that's the same as the topic that you specified.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM '/company/windfarm/+/turbine/+/+'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "iotSiteWise": {
                    "putAssetPropertyValueEntries": [
                        {
                            "propertyAlias": "${topic()}",
                            "propertyValues": [
                                {
                                    "timestamp": {
                                        "timeInSeconds": "${floor(timestamp() / 1E3)}",
                                        "offsetInNanos": "${(timestamp() % 1E3) * 1E6}"
                                    },
                                    "value": {
                                        "doubleValue": "${my.payload.value}"
                                    }
                                }
                            ]
                        }
                    ],
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sitewise"
                }
            }
        ]
    }
}
```

See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*

- Ingesting data using AWS IoT Core rules in the *AWS IoT SiteWise User Guide*
- Ingesting data to AWS IoT SiteWise from AWS IoT things in the *AWS IoT SiteWise User Guide*
- Troubleshooting an AWS IoT SiteWise rule action in the *AWS IoT SiteWise User Guide*

Firehose

The Firehose(`firehose`) action sends data from an MQTT message to an Amazon Data Firehose stream.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `firehose:PutRecord` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use Firehose to send data to an Amazon S3 bucket, and you use an AWS KMS customer managed AWS KMS key to encrypt data at rest in Amazon S3, Firehose must have access to your bucket and permission to use the AWS KMS key on the caller's behalf. For more information, see Grant Firehose access to an Amazon S3 destination in the *Amazon Data Firehose Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`batchMode`

(Optional) Whether to deliver the Firehose stream as a batch by using `PutRecordBatch` . The default value is `false`.

When `batchMode` is `true` and the rule's SQL statement evaluates to an Array, each Array element forms one record in the `PutRecordBatch` request. The resulting array can't have more than 500 records.

Supports substitution templates: No

`deliveryStreamName`

The Firehose stream to which to write the message data.

Supports substitution templates: API and AWS CLI only

`separator`

(Optional) A character separator that is used to separate records written to the Firehose stream. If you omit this parameter, the stream uses no separator. Valid values: `,` (comma), `\t` (tab), `\n` (newline), `\r\n` (Windows newline).

Supports substitution templates: No

`roleArn`

The IAM role that allows access to the Firehose stream. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a Firehose action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "firehose": {
                    "deliveryStreamName": "my_firehose_stream",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose"
                }
            }
        ]
    }
}
```

The following JSON example defines a Firehose action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "firehose": {
                    "deliveryStreamName": "${topic()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose"
                }
            }
        ]
    }
}
```

See also

- What is Amazon Data Firehose? in the *Amazon Data Firehose Developer Guide*

Kinesis Data Streams

The Kinesis Data Streams (`kinesis`) action writes data from an MQTT message to Amazon Kinesis Data Streams.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `kinesis:PutRecord` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use an AWS KMS customer-managed AWS KMS key (KMS key) to encrypt data at rest in Kinesis Data Streams, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see Permissions to use user-generated AWS KMS keys in the *Amazon Kinesis Data Streams Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`stream`

The Kinesis data stream to which to write data.

Supports substitution templates: API and AWS CLI only

`partitionKey`

The partition key used to determine to which shard the data is written. The partition key is usually composed of an expression (for example, `${topic()}` or `${timestamp()}`).

Supports substitution templates: Yes

`roleArn`

The ARN of the IAM role that grants AWS IoT permission to access the Kinesis data stream. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a Kinesis Data Streams action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "kinesis": {
                    "streamName": "my_kinesis_stream",
                    "partitionKey": "${topic()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_kinesis"
                }
            }
        ]
    }
```

```
}
```

The following JSON example defines a Kinesis action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "kinesis": {
                    "streamName": "${topic()}",
                    "partitionKey": "${timestamp()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_kinesis"
                }
            }
        ]
    }
}
```

See also

- What is Amazon Kinesis Data Streams? in the *Amazon Kinesis Data Streams Developer Guide*

Lambda

A Lambda (`lambda`) action invokes an AWS Lambda function, passing in an MQTT message. AWS IoT invokes Lambda functions asynchronously.

You can follow a tutorial that shows you how to create and test a rule with a Lambda action. For more information, see Tutorial: Formatting a notification by using an AWS Lambda function.

Requirements

This rule action has the following requirements:

- For AWS IoT to invoke a Lambda function, you must configure a policy that grants the `lambda:InvokeFunction` permission to AWS IoT. You can only invoke a Lambda function defined in the same AWS Region where your Lambda policy exists. Lambda functions use resource-based policies, so you must attach the policy to the Lambda function itself.

Use the following AWS CLI command to attach a policy that grants the `lambda:InvokeFunction` permission. In this command, replace:

- *`function_name`* with the name of the Lambda function. You add a new permission to update the function's resource policy.
- *`region`* with the AWS Region of the function.
- *`account-id`* with the AWS account number where the rule is defined.
- *`rule-name`* with the name of the AWS IoT rule for which you are defining the Lambda action.
- *`unique_id`* with a unique statement identifier.

> **Important**
>
> If you add a permission for an AWS IoT principal without providing the `source-arn` or `source-account`, any AWS account that creates a rule with your Lambda action can activate rules to invoke your Lambda function from AWS IoT.

For more information, see AWS Lambda permissions.

```
aws lambda add-permission \
    --function-name function_name \
    --region region \
    --principal iot.amazonaws.com \
    --source-arn arn:aws:iot:region:account-id:rule/rule_name \
    --source-account account-id
    --statement-id unique_id
    --action "lambda:InvokeFunction"
```

- If you use the AWS IoT console to create a rule for the Lambda rule action, the Lambda function is triggered automatically. If you use AWS CloudFormation instead with the `AWS::IoT::TopicRule LambdaAction`, you must add an `AWS::lambda::Permission` resource. The resource then grants you permission to trigger the Lambda function.

  The following code shows an example of how to add this resource. In this example, replace:

  - *`function_name`* with the name of the Lambda function.
  - *`region`* with the AWS Region of the function.

- *account-id* with the AWS account number where the rule is defined.
- *rule-name* with the name of the AWS IoT rule for which you are defining the Lambda action.

```
Type: AWS::Lambda::Permission
Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref function_name
  Principal: "iot.amazonaws.com"
  SourceAccount: account-id
  SourceArn: arn:aws:iot:region:account-id:rule/rule_name
```

- If you use an AWS KMS customer managed AWS KMS key to encrypt data at rest in Lambda, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see Encryption at rest in the *AWS Lambda Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`functionArn`

The ARN of the Lambda function to invoke. AWS IoT must have permission to invoke the function. For more information, see Requirements.

If you don't specify a version or alias for your Lambda function, the most recent version of the function is shut down. You can specify a version or alias if you want to shut down a specific version of your Lambda function. To specify a version or alias, append the version or alias to the ARN of the Lambda function.

```
arn:aws:lambda:us-east-2:123456789012:function:myLambdaFunction:someAlias
```

For more information about versioning and aliases, and see AWS Lambda function versioning and aliases.

Supports substitution templates: API and AWS CLI only

Examples

The following JSON example defines a Lambda action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "lambda": {
                    "functionArn": "arn:aws:lambda:us-
east-2:123456789012:function:myLambdaFunction"
                 }
            }
        ]
    }
}
```

The following JSON example defines a Lambda action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "lambda": {
                    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:
${topic()}"
                }
            }
        ]
    }
}
```

See also

- What is AWS Lambda? in the *AWS Lambda Developer Guide*
- Tutorial: Formatting a notification by using an AWS Lambda function

Location

The Location (`location`) action sends your geographical location data to Amazon Location Service.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `geo:BatchUpdateDevicePosition` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`deviceId`

The unique ID of the device providing the location data. For more information, see `DeviceId` from the *Amazon Location Service API Reference*.

Supports substitution templates: Yes

`latitude`

A string that evaluates to a double value that represents the latitude of the device's location.

Supports substitution templates: Yes

`longitude`

A string that evaluates to a double value that represents the longitude of the device's location.

Supports substitution templates: Yes

`roleArn`

The IAM role that allows access to the Amazon Location Service domain. For more information, see Requirements.

`timestamp`

The time that the location data was sampled. The default value is the time that the MQTT message was processed.

The `timestamp` value consists of the following two values:

- `value`: An expression that returns a long epoch time value. You can use the the section called "time_to_epoch(String, String)" function to create a valid timestamp from a date or time value passed in the message payload. Supports substitution templates: Yes.
- `unit`: (Optional) The precision of the timestamp value that results from the expression described in `value`. Valid values: `SECONDS` | `MILLISECONDS` | `MICROSECONDS` | `NANOSECONDS`. The default is `MILLISECONDS`. Supports substitution templates: API and AWS CLI only.

`trackerName`

The name of the tracker resource in Amazon Location in which the location is updated. For more information, see Tracker from the *Amazon Location Service Developer Guide*.

Supports substitution templates: API and AWS CLI only

Examples

The following JSON example defines a Location action in an AWS IoT rule.

```
{
 "topicRulePayload": {
  "sql": "SELECT * FROM 'some/topic'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
   {
    "location": {
     "roleArn": "arn:aws:iam::123454962127:role/service-role/ExampleRole",
     "trackerName": "MyTracker",
     "deviceId": "001",
     "sampleTime": {
      "value": "${timestamp()}",
      "unit": "MILLISECONDS"
     },
```

```
      "latitude": "-12.3456",
      "longitude": "65.4321"
     }
    }
   ]
  }
}
```

The following JSON example defines a Location action with substitution templates in an AWS IoT rule.

```
{
 "topicRulePayload": {
  "sql": "SELECT * FROM 'some/topic'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
   {
    "location": {
     "roleArn": "arn:aws:iam::123456789012:role/service-role/ExampleRole",
     "trackerName": "${TrackerName}",
     "deviceId": "${DeviceID}",
     "timestamp": {
      "value": "${timestamp()}",
      "unit": "MILLISECONDS"
     },
     "latitude": "${get(position, 0)}",
     "longitude": "${get(position, 1)}"
    }
   }
  ]
 }
}
```

The following MQTT payload example shows how substitution templates in the preceding example accesses data. You can use the [get-device-position-history](https://docs.aws.amazon.com/cli/latest/reference/location/get-device-position-history.html) CLI command to verify that the MQTT payload data is delivered in your location tracker.

```
{
 "TrackerName": "mytracker",
 "DeviceID": "001",
 "position": [
```

```
    "-12.3456",
    "65.4321"
  ]
}
```

```
aws location get-device-position-history --device-id 001 --tracker-name mytracker
```

```
{
  "DevicePositions": [
   {
    "DeviceId": "001",
    "Position": [
     -12.3456,
     65.4321
    ],
    "ReceivedTime": "2022-11-11T01:31:54.464000+00:00",
    "SampleTime": "2022-11-11T01:31:54.308000+00:00"
   }
  ]
}
```

See also

- What is Amazon Location Service? in the *Amazon Location Service Developer Guide*.

OpenSearch

The OpenSearch (`openSearch`) action writes data from MQTT messages to an Amazon OpenSearch Service domain. You can then use tools like OpenSearch Dashboards to query and visualize data in OpenSearch Service.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `es:ESHttpPut` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use a customer managed AWS KMS key to encrypt data at rest in OpenSearch Service, the service must have permission to use the KMS key on the caller's behalf. For more information, see Encryption of data at rest for Amazon OpenSearch Service in the *Amazon OpenSearch Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`endpoint`

The endpoint of your Amazon OpenSearch Service domain.

Supports substitution templates: API and AWS CLI only

`index`

The OpenSearch index where you want to store your data.

Supports substitution templates: Yes

`type`

The type of document you are storing.

> **Note**
>
> For OpenSearch versions later than 1.0, the value of the `type` parameter must be `_doc`. For more information, see the OpenSearch documentation.

Supports substitution templates: Yes

`id`

The unique identifier for each document.

Supports substitution templates: Yes

`roleARN`

The IAM role that allows access to the OpenSearch Service domain. For more information, see Requirements.

Supports substitution templates: No

Limitations

The OpenSearch (`openSearch`) action cannot be used to deliver data to VPC Elasticsearch clusters.

Examples

The following JSON example defines an OpenSearch action in an AWS IoT rule and how you can specify the fields for the `OpenSearch` action. For more information, see OpenSearchAction.

```
{
    "topicRulePayload": {
        "sql": "SELECT *, timestamp() as timestamp FROM 'iot/test'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "openSearch": {
                    "endpoint": "https://my-endpoint",
                    "index": "my-index",
                    "type": "_doc",
                    "id": "${newuuid()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_os"
                }
            }
        ]
    }
}
```

The following JSON example defines an OpenSearch action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "openSearch": {
                    "endpoint": "https://my-endpoint",
                    "index": "${topic()}",
                    "type": "${type}",
```

```
                    "id": "${newuuid()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_os"
                }
            }
        ]
    }
}
```

> **Note**
>
> The the substituted `type` field works for OpenSearch version 1.0. For any versions later than 1.0, the value of `type` must be `_doc`.

See also

What is Amazon OpenSearch Service? in the *Amazon OpenSearch Service Developer Guide*

Republish

The republish (`republish`) action republishes an MQTT message to another MQTT topic.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `iot:Publish` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`headers`

MQTT Version 5.0 headers information.

For more information, see RepublishAction and MqttHeaders in the *AWS API Reference*.

`topic`

The MQTT topic to which to republish the message.

To republish to a reserved topic, which begins with `$`, use `$$` instead. For example, to republish to the device shadow topic `$aws/things/MyThing/shadow/update`, specify the topic as `$$aws/things/MyThing/shadow/update`.

> **Note**
>
> Republishing to reserved job topics is not supported.
> AWS IoT Device Defender reserve topics don't support HTTP publish.

Supports substitution templates: Yes

`qos`

(Optional) The Quality of Service (QoS) level to use when republishing messages. Valid values: `0`, `1`. The default value is `0`. For more information about MQTT QoS, see MQTT.

Supports substitution templates: No

`roleArn`

The IAM role that allows AWS IoT to publish to the MQTT topic. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a republish action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "republish": {
```

```
                    "topic": "another/topic",
                    "qos": 1,
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish"
                }
            }
        ]
    }
}
```

The following JSON example defines a republish action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "republish": {
                    "topic": "${topic()}/republish",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish"
                }
            }
        ]
    }
}
```

The following JSON example defines a republish action with `headers` in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "republish": {
                    "topic": "${topic()}/republish",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish",
                    "headers": {
                        "payloadFormatIndicator": "UTF8_DATA",
                        "contentType": "rule/contentType",
```

```
                                "correlationData": "cnVsZSBjb3JyZWxhdGlvbiBkYXRh",
                                "userProperties": [
                                    {
                                        "key": "ruleKey1",
                                        "value": "ruleValue1"
                                    },
                                    {
                                        "key": "ruleKey2",
                                        "value": "ruleValue2"
                                    }
                                ]
                            }
                        }
                    }
                ]
            }
        }
```

Note

The original source IP won't be passed though Republish action.

S3

The S3 (`s3`) action writes the data from an MQTT message to an Amazon Simple Storage Service (Amazon S3) bucket.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `s3:PutObject` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

- If you use an AWS KMS customermanaged AWS KMS key to encrypt data at rest in Amazon S3, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see AWS managed AWS KMS keys and customer managed AWS KMS keys in the *Amazon Simple Storage Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`bucket`

The Amazon S3 bucket to which to write data.

Supports substitution templates: API and AWS CLI only

`cannedacl`

(Optional) The Amazon S3 canned ACL that controls access to the object identified by the object key. For more information, including allowed values, see Canned ACL.

Supports substitution templates: No

`key`

The path to the file where the data is written.

Consider an example where this parameter is `${topic()}/${timestamp()}` and the rule receives a message where the topic is `some/topic`. If the current timestamp is `1460685389`, then this action writes the data to a file called `1460685389` in the `some/topic` folder of the S3 bucket.

> **Note**
>
> If you use a static key, AWS IoT overwrites a single file each time the rule invokes. We recommend that you use the message timestamp or another unique message identifier so that a new file is saved in Amazon S3 for each message received.

Supports substitution templates: Yes

`roleArn`

The IAM role that allows access to the Amazon S3 bucket. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines an S3 action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "s3": {
                    "bucketName": "amzn-s3-demo-bucket",
                    "cannedacl": "public-read",
                    "key": "${topic()}/${timestamp()}",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3"
                }
            }
        ]
    }
}
```

See also

- [What is Amazon S3?](#) in the *Amazon Simple Storage Service User Guide*

Salesforce IoT

The Salesforce IoT (`salesforce`) action sends data from the MQTT message that triggered the rule to a Salesforce IoT input stream.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`url`

The URL exposed by the Salesforce IoT input stream. The URL is available from the Salesforce IoT platform when you create an input stream. For more information, see the Salesforce IoT documentation.

Supports substitution templates: No

`token`

The token used to authenticate access to the specified Salesforce IoT input stream. The token is available from the Salesforce IoT platform when you create an input stream. For more information, see the Salesforce IoT documentation.

Supports substitution templates: No

Examples

The following JSON example defines a Salesforce IoT action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "salesforce": {
                    "token": "ABCDEFGHI123456789abcdefghi123456789",
                    "url": "https://ingestion-cluster-id.my-env.sfdcnow.com/streams/
stream-id/connection-id/my-event"
                }
            }
        ]
    }
}
```

SNS

The SNS (`sns`) action sends the data from an MQTT message as an Amazon Simple Notification Service (Amazon SNS) push notification.

You can follow a tutorial that shows you how to create and test a rule with an SNS action. For more information, see Tutorial: Sending an Amazon SNS notification.

> **Note**
>
> The SNS action doesn't support Amazon SNS FIFO (First-In-First-Out) topics. Because the rules engine is a fully distributed service, there is no guarantee of message order when the SNS action is invoked.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `sns:Publish` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use an AWS KMS customer managed-managed AWS KMS key to encrypt data at rest in Amazon SNS, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see Key management in the *Amazon Simple Notification Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`targetArn`

The SNS topic or individual device to which the push notification is sent.

Supports substitution templates: API and AWS CLI only

`messageFormat`

(Optional) The message format. Amazon SNS uses this setting to determine if the payload should be parsed and if relevant platform-specific parts of the payload should be extracted. Valid values: `JSON`, `RAW`. Defaults to `RAW`.

Supports substitution templates: No

`roleArn`

The IAM role that allows access to SNS. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines an SNS action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "sns": {
                    "targetArn": "arn:aws:sns:us-east-2:123456789012:my_sns_topic",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sns"
                }
            }
        ]
    }
}
```

The following JSON example defines an SNS action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "sns": {
                    "targetArn": "arn:aws:sns:us-east-1:123456789012:${topic()}",
                    "messageFormat": "JSON",
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sns"
                }
            }
        ]
    }
}
```

See also

- What is Amazon Simple Notification Service? in the *Amazon Simple Notification Service Developer Guide*
- Tutorial: Sending an Amazon SNS notification

SQS

The SQS (`sqs`) action sends data from an MQTT message to an Amazon Simple Queue Service (Amazon SQS) queue.

> **Note**
>
> The SQS action doesn't support Amazon SQS FIFO (First-In-First-Out) queues. Because the rules engine is a fully distributed service, there is no guarantee of message order when the SQS action is triggered.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `sqs:SendMessage` operation. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.
- If you use an AWS KMS customer managed AWS KMS key to encrypt data at rest in Amazon SQS, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see Key management in the *Amazon Simple Queue Service Developer Guide*.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`queueUrl`

The URL of the Amazon SQS queue to which to write the data. The region in this URL doesn't need to be the same AWS Region as your AWS IoT rule.

> **Note**
>
> There can be additional charges for data transfer cross AWS Regions using the SQS rule action. For more information, see Amazon SQS pricing.

Supports substitution templates: API and AWS CLI only

`useBase64`

Set this parameter to `true` to configure the rule action to base64-encode the message data before it writes the data to the Amazon SQS queue. Defaults to `false`.

Supports substitution templates: No

`roleArn`

The IAM role that allows access to the Amazon SQS queue. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines an SQS action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "sqs": {
                    "queueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/
my_sqs_queue",
```

```
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sqs"
                }
            }
        ]
    }
}
```

The following JSON example defines an SQS action with substitution templates in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "sqs": {
                    "queueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/
${topic()}",
                    "useBase64": true,
                    "roleArn": "arn:aws:iam::123456789012:role/aws_iot_sqs"
                }
            }
        ]
    }
}
```

See also

- What is Amazon Simple Queue Service? in the *Amazon Simple Queue Service Developer Guide*

Step Functions

The Step Functions (`stepFunctions`) action starts an AWS Step Functions state machine.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `states:StartExecution` operation. For more information, see Granting an AWS IoT rule the access it requires.

In the AWS IoT console, you can choose or create a role to allow AWS IoT to perform this rule action.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`stateMachineName`

The name of the Step Functions state machine to start.

Supports substitution templates: API and AWS CLI only

`executionNamePrefix`

(Optional) The name given to the state machine execution consists of this prefix followed by a UUID. Step Functions creates a unique name for each state machine execution if one is not provided.

Supports substitution templates: Yes

`roleArn`

The ARN of the role that grants AWS IoT permission to start the state machine. For more information, see Requirements.

Supports substitution templates: No

Examples

The following JSON example defines a Step Functions action in an AWS IoT rule.

```
{
    "topicRulePayload": {
        "sql": "SELECT * FROM 'some/topic'",
        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "stepFunctions": {
                    "stateMachineName": "myStateMachine",
                    "executionNamePrefix": "myExecution",
```

```
                     "roleArn": "arn:aws:iam::123456789012:role/aws_iot_step_functions"
                 }
             }
         ]
     }
 }
```

See also

- What is AWS Step Functions? in the *AWS Step Functions Developer Guide*

Timestream

The Timestream rule action writes attributes (measures) from an MQTT message into an Amazon Timestream table. For more information about Amazon Timestream, see What Is Amazon Timestream?.

> **Note**
>
> Amazon Timestream is not available in all AWS Regions. If Amazon Timestream is not available in your Region, it won't appear in the list of rule actions.

The attributes that this rule stores in the Timestream database are those that result from the rule's query statement. The value of each attribute in the query statement's result is parsed to infer its data type (as in a the section called “DynamoDBv2” action). Each attribute's value is written to its own record in the Timestream table. To specify or change an attribute's data type, use the `cast()` function in the query statement. For more information about the contents of each Timestream record, see the section called “Timestream record content”.

> **Note**
>
> With SQL V2 (2016-03-23), numeric values that are whole numbers, such as `10.0`, are converted their Integer representation (`10`). Explicitly casting them to a `Decimal` value, such as by using the cast() function, does not prevent this behavior—the result is still an `Integer` value. This can cause type mismatch errors that prevent data from being recorded in the Timestream database. To process whole number numeric values as `Decimal` values, use SQL V1 (2015-10-08) for the rule query statement.

> **Note**
>
> The maximum number of values that a Timestream rule action can write into an Amazon Timestream table is 100. For more information, see Amazon Timestream Quota's Reference.

Requirements

This rule action has the following requirements:

- An IAM role that AWS IoT can assume to perform the `timestream:DescribeEndpoints` and `timestream:WriteRecords` operations. For more information, see Granting an AWS IoT rule the access it requires.

  In the AWS IoT console, you can choose, update, or create a role to allow AWS IoT to perform this rule action.
- If you use a customer- AWS KMS to encrypt data at rest in Timestream, the service must have permission to use the AWS KMS key on the caller's behalf. For more information, see How AWS services use AWS KMS.

Parameters

When you create an AWS IoT rule with this action, you must specify the following information:

`databaseName`

The name of an Amazon Timestream database that has the table to receive the records this action creates. See also **`tableName`**.

Supports substitution templates: API and AWS CLI only

`dimensions`

Metadata attributes of the time series that are written in each measure record. For example, the name and Availability Zone of an EC2 instance or the name of the manufacturer of a wind turbine are dimensions.

`name`

The metadata dimension name. This is the name of the column in the database table record.

Dimensions can't be named: `measure_name`, `measure_value`, or `time`. These names are reserved. Dimension names can't start with `ts_` or `measure_value` and they can't contain the colon (`:`) character.

Supports substitution templates: No

`value`

The value to write in this column of the database record.

Supports substitution templates: Yes

`roleArn`

The Amazon Resource Name (ARN) of the role that grants AWS IoT permission to write to the Timestream database table. For more information, see Requirements.

Supports substitution templates: No

`tableName`

The name of the database table into which to write the measure records. See also **`databaseName`**.

Supports substitution templates: API and AWS CLI only

`timestamp`

The value to use for the entry's timestamp. If blank, the time that the entry was processed is used.

`unit`

The precision of the timestamp value that results from the expression described in `value`.

Valid values: `SECONDS` | `MILLISECONDS` | `MICROSECONDS` | `NANOSECONDS`. The default is `MILLISECONDS`.

`value`

An expression that returns a long epoch time value.

You can use the the section called "time_to_epoch(String, String)" function to create a valid timestamp from a date or time value passed in the message payload.

Timestream record content

The data written to the Amazon Timestream table by this action include a timestamp, metadata from the Timestream rule action, and the result of the rule's query statement.

For each attribute (measure) in the result of the query statement, this rule action writes a record to the specified Timestream table with these columns.

Column name	Attribute type	Value	Comments
dimension-name	DIMENSION	The value specified in the Timestream rule action entry.	Each **Dimension** specified in the rule action entry creates a column in the Timestream database with the dimension's name.
measure_name	MEASURE_NAME	The attribute's name	The name of the attribute in the result of the query statement whose value is specified in the `measure_value:: data-type` column.
measure_value::*data-type*	MEASURE_VALUE	The value of the attribute in the result of the query statement. The attribute's name is in the `measure_name` column.	The value is interpreted* and cast as the most suitable match of: `bigint`, `boolean`, `double`, or `varchar`. Amazon Timestream creates a separate column for each data type. The value in the message can be cast

Column name	Attribute type	Value	Comments
			to another data type by using the cast() function in the rule's query statement.
time	TIMESTAMP	The date and time of the record in the database.	This value is assigned by rules engine or the `timestamp` property, if it is defined.

* The attribute value read from the message payload is interpreted as follows. See the the section called "Examples" for an illustration of each of these cases.

- An unquoted value of `true` or `false` is interpreted as a `boolean` type.
- A decimal numeric is interpreted as a `double` type.
- A numeric value without a decimal point is interpreted as a `bigint` type.
- A quoted string is interpreted as a `varchar` type.
- Objects and array values are converted to JSON strings and stored as a `varchar` type.

Examples

The following JSON example defines a Timestream rule action with a substitution template in an AWS IoT rule.

```
{
  "topicRulePayload": {
    "sql": "SELECT * FROM 'iot/topic'",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "timestream": {
          "roleArn": "arn:aws:iam::123456789012:role/aws_iot_timestream",
          "tableName": "devices_metrics",
          "dimensions": [
```

```
            {
              "name": "device_id",
              "value": "${clientId()}"
            },
            {
              "name": "device_firmware_sku",
              "value": "My Static Metadata"
            }
          ],
          "databaseName": "record_devices"
        }
      }
    ]
  }
}
```

Using the Timestream topic rule action defined in the previous example with the following message payload results in the Amazon Timestream records written in the table that follows.

```
{
  "boolean_value": true,
  "integer_value": 123456789012,
  "double_value": 123.456789012,
  "string_value": "String value",
  "boolean_value_as_string": "true",
  "integer_value_as_string": "123456789012",
  "double_value_as_string": "123.456789012",
  "array_of_integers": [23,36,56,72],
  "array of strings": ["red", "green","blue"],
  "complex_value": {
    "simple_element": 42,
    "array_of_integers": [23,36,56,72],
    "array of strings": ["red", "green","blue"]
  }
}
```

The following table displays the database columns and records that using the specified topic rule action to process the previous message payload creates. The `device_firmware_sku` and `device_id` columns are the DIMENSIONS defined in the topic rule action. The Timestream topic rule action creates the `time` column and the `measure_name` and `measure_value::*` columns, which it fills with the values from the result of the topic rule action's query statement.

device_firmware_sku	device_id	measure_name	measure_value::bigint	measure_value::varchar	measure_value::double	measure_value::boolean	time
My Static Metadata	iotconsole-159EXAMPLE738-0	complex_value	-	{"simple_element":42,"array_of_integers":[23,36,56,72],"array of strings":["red","green","blue"]}	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	integer_value_as_string	-	123456789012	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	boolean_value	-	-	-	TRUE	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	integer_value	123456789012	-	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	string_value	-	String value	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	array_of_integers	-	[23,36,56,72]	-	-	2020-08-26 22:42:16.423000000

device_firmware_sku	device_id	measure_name	measure_value::bigint	measure_value::varchar	measure_value::double	measure_value::boolean	time
My Static Metadata	iotconsole-159EXAMPLE738-0	array of strings	-	["red","green","blue"]	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	boolean_value_as_string	-	TRUE	-	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	double_value	-	-	123.456789012	-	2020-08-26 22:42:16.423000000
My Static Metadata	iotconsole-159EXAMPLE738-0	double_value_as_string	-	123.45679	-	-	2020-08-26 22:42:16.423000000

Troubleshooting a rule

If you have an issue with your rules, we recommend that you activate CloudWatch Logs. You can analyze your logs to determine whether the issue is authorization or whether, for example, a WHERE clause condition didn't match. For more information, see Setting Up CloudWatch Logs.

Accessing cross-account resources using AWS IoT rules

You can configure AWS IoT rules for cross-account access so that data ingested on MQTT topics of one account can be routed into the AWS services, such as Amazon SQS and Lambda, of another account. The following explains how to set up AWS IoT rules for cross-account data ingestion, from an MQTT topic in one account, to a destination in another account.

Cross-account rules can be configured using resource-based permissions on the destination resource. Therefore, only destinations that support resource-based permissions can be enabled for the cross-account access with AWS IoT rules. The supported destinations include Amazon SQS, Amazon SNS, Amazon S3, and AWS Lambda.

> **Note**
>
> For the supported destinations, except for Amazon SQS, you must define the rule in the same AWS Region as another service's resource so that the rule action can interact with that resource. For more information about AWS IoT rule actions, see AWS IoT rule actions. For more information about rule's SQS action, see ???.

Prerequisites

- Familiarity with AWS IoT rules
- An understanding of IAM users, roles, and resource-based permission
- Having AWS CLI installed

Cross-account setup for Amazon SQS

Scenario: Account A sends data from an MQTT message to account B's Amazon SQS queue.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: `sqs:SendMessage`
2222-2222-2222	Account B	Amazon SQS queue • ARN: *arn:aws:sqs:region:2222-2222-2222:ExampleQueue* • URL: *https://sqs.region.amazonaws.com/2222-2222-2222/ExampleQueue*

> **Note**
>
> Your destination Amazon SQS queue doesn't have to be in the same AWS Region as your AWS IoT rule. For more information about rule's SQS action, see ???.

Do the Account A tasks

> **Note**
>
> To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with the rule's Amazon Resource Name (ARN) as a resource, and permissions to `iam:PassRole` action with a resource as the role's ARN.

1. Configure AWS CLI using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine, and attaches a policy that allows access to account B's Amazon SQS queue. See example commands and policy documents in Granting AWS IoT the required access.
3. To create a rule that is attached to a topic, run the create-topic-rule command.

   ```
   aws iot create-topic-rule --rule-name myRule --topic-rule-payload file://./my-rule.json
   ```

   The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon SQS queue. The SQL statement filters the messages and the role ARN grants AWS IoT permissions to add the message to the Amazon SQS queue.

   ```
   {
    "sql": "SELECT * FROM 'iot/test'",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
     {
      "sqs": {
       "queueUrl": "https://sqs.region.amazonaws.com/2222-2222-2222/ExampleQueue",
       "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role",
       "useBase64": false
      }
   ```

```
    }
  ]
}
```

For more information about how to define an Amazon SQS action in an AWS IoT rule, see AWS IoT rule actions - Amazon SQS.

Do the Account B tasks

1. Configure AWS CLI using account B's IAM user.
2. To give permissions for the Amazon SQS queue resource to account A, run the add-permission command.

```
aws sqs add-permission --queue-url https://sqs.region.amazonaws.com/2222-2222-2222/
ExampleQueue --label SendMessagesToMyQueue --aws-account-ids 1111-1111-1111 --
actions SendMessage
```

Cross-account setup for Amazon SNS

Scenario: Account A sends data from an MQTT message to an Amazon SNS topic of account B.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: `sns:Publish`
2222-2222-2222	Account B	Amazon SNS topic ARN: *arn:aws:sns:region:2222-2222-2222:ExampleTopic*

Do the Account A tasks

> **Notes**
>
> To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with rule ARN as a resource and permissions to the `iam:PassRole` action with a resource as role ARN.

1. Configure AWS CLI using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine, and attaches a policy that allows access to account B's Amazon SNS topic. For example commands and policy documents, see Granting AWS IoT the required access.
3. To create a rule that is attached to a topic, run the create-topic-rule command.

```
aws iot create-topic-rule --rule-name myRule --topic-rule-payload file://./my-rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon SNS topic. The SQL statement filters the messages, and the role ARN grants AWS IoT permissions to send the message to the Amazon SNS topic.

```
{
 "sql": "SELECT * FROM 'iot/test'",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "sns": {
    "targetArn": "arn:aws:sns:region:2222-2222-2222:ExampleTopic",
    "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role"
   }
  }
 ]
}
```

For more information about how to define an Amazon SNS action in an AWS IoT rule, see AWS IoT rule actions - Amazon SNS.

Do the Account B tasks

1. Configure AWS CLI using account B's IAM user.
2. To give permission on the Amazon SNS topic resource to account A, run the add-permission command.

```
aws sns add-permission --topic-arn arn:aws:sns:region:2222-2222-2222:ExampleTopic
 --label Publish-Permission --aws-account-id 1111-1111-1111 --action-name Publish
```

Cross-account setup for Amazon S3

Scenario: Account A sends data from an MQTT message to an Amazon S3 bucket of account B.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: `s3:PutObject`
2222-2222-2222	Account B	Amazon S3 bucket ARN: *arn:aws:s3:::amzn-s3-demo-bucket*

Do the Account A tasks

Note

To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with the rule ARN as a resource and permissions to `iam:PassRole` action with a resource as role ARN.

1. Configure AWS CLI using account A's IAM user.
2. Create an IAM role that trusts AWS IoT rules engine and attaches a policy that allows access to account B's Amazon S3 bucket. For example commands and policy documents, see Granting AWS IoT the required access.
3. To create a rule that is attached to your target S3 bucket, run the create-topic-rule command.

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://./my-rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Amazon S3 bucket. The SQL statement filters the messages, and the role ARN grants AWS IoT permissions to add the message to the Amazon S3 bucket.

```
{
 "sql": "SELECT * FROM 'iot/test'",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "s3": {
    "bucketName": "amzn-s3-demo-bucket",
    "key": "${topic()}/${timestamp()}",
    "roleArn": "arn:aws:iam::1111-1111-1111:role/my-iot-role"
   }
  }
 ]
}
```

For more information about how to define an Amazon S3 action in an AWS IoT rule, see [AWS IoT rule actions - Amazon S3](#).

Do the Account B tasks

1. Configure AWS CLI using account B's IAM user.
2. Create a bucket policy that trusts account A's principal.

   The following is an example payload file that defines a bucket policy that trusts the principal of another account.

   ```
   {
       "Version":"2012-10-17",
       "Statement": [
           {
               "Sid": "AddCannedAcl",
   ```

```
                "Effect": "Allow",
                "Principal": {
                    "AWS": [
                        "arn:aws:iam::123456789012:root"
                    ]
            },
                "Action": "s3:PutObject",
                "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
            }
        ]
}
```

For more information, see bucket policy examples.

3. To attach the bucket policy to the specified bucket, run the put-bucket-policy command.

```
aws s3api put-bucket-policy --bucket amzn-s3-demo-bucket --policy file://./amzn-s3-demo-bucket-policy.json
```

4. To make the cross-account access work, make sure you have the correct **Block all public access** settings. For more information, see Security Best Practices for Amazon S3.

Cross-account setup for AWS Lambda

Scenario: Account A invokes an AWS Lambda function of account B, passing in an MQTT message.

AWS account	Account referred to as	Description
1111-1111-1111	Account A	Rule action: `lambda:InvokeFunction`
2222-2222-2222	Account B	Lambda function ARN: *arn:aws:lambda:region:2222-2222-2222:function:example-function*

Do the Account A tasks

> **Notes**
>
> To run the following commands, your IAM user should have permissions to `iot:CreateTopicRule` with rule ARN as a resource, and permissions to `iam:PassRole` action with resource as role ARN.

1. Configure AWS CLI using account A's IAM user.
2. Run the create-topic-rule command to create a rule that defines cross-account access to account B's Lambda function.

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://./my-rule.json
```

   The following is an example payload file with a rule that inserts all messages sent to the `iot/test` topic into the specified Lambda function. The SQL statement filters the messages and the role ARN grants AWS IoT permission to pass in the data to the Lambda function.

```
{
 "sql": "SELECT * FROM 'iot/test'",
 "ruleDisabled": false,
 "awsIotSqlVersion": "2016-03-23",
 "actions": [
  {
   "lambda": {
    "functionArn": "arn:aws:lambda:region:2222-2222-2222:function:example-function"
   }
  }
 ]
}
```

   For more information about how to define an AWS Lambda action in an AWS IoT rule, read AWS IoT rule actions - Lambda.

Do the Account B tasks

1. Configure AWS CLI using account B's IAM user.

2. Run [Lambda's add-permission command](#) to give AWS IoT rules permission to activate the Lambda function. To run the following command, your IAM user should have permission to `lambda:AddPermission` action.

```
aws lambda add-permission --function-name example-function --region us-east-1 --
principal iot.amazonaws.com --source-arn arn:aws:iot:region:1111-1111-1111:rule/
example-rule --source-account 1111-1111-1111 --statement-id "unique_id" --action
 "lambda:InvokeFunction"
```

Options:

--principal

This field gives permission to AWS IoT (represented by `iot.amazonaws.com`) to call the Lambda function.

--source-arn

This field confirms that only `arn:aws:iot:region:1111-1111-1111:rule/example-rule` in AWS IoT triggers this Lambda function and no other rule in the same or different account can activate this Lambda function.

--source-account

This field confirms that AWS IoT activates this Lambda function only on behalf of the `1111-1111-1111` account.

Notes

If you see an error message "The rule could not be found" from your AWS Lambda function's console under **Configuration**, ignore the error message and proceed to test the connection.

Error handling (error action)

When AWS IoT receives a message from a device, the rules engine checks to see if the message matches a rule. If so, the rule's query statement is evaluated and the rule's actions are activated, passing the query statement's result.

If a problem occurs when activating an action, the rules engine activates an error action, if one is specified for the rule. This might happen when:

- A rule doesn't have permission to access an Amazon S3 bucket.
- A user error causes DynamoDB provisioned throughput to be exceeded.

> **Note**
>
> The error handling covered in this topic is for rule actions. To debug SQL issues, including external functions, you can set up AWS IoT logging. For more information, see ???.

Error action message format

A single message is generated per rule and message. For example, if two rule actions in the same rule fail, the error action receives one message that contains both errors.

The error action message looks like the following example.

```
{
  "ruleName": "TestAction",
  "topic": "testme/action",
  "cloudwatchTraceId": "7e146a2c-95b5-6caf-98b9-50e3969734c7",
  "clientId": "iotconsole-1511213971966-0",
  "base64OriginalPayload":
 "ewogICJtZXNzYWdlIjogIkhlbGxvIHZyb20gQVdTIElvVCBjb25zb2xlIgp9",
  "failures": [
    {
      "failedAction": "S3Action",
      "failedResource": "us-east-1-s3-verify-user",
      "errorMessage": "Failed to put S3 object. The error received was The
 specified bucket does not exist (Service: Amazon S3; Status Code: 404; Error
 Code: NoSuchBucket; Request ID: 9DF5416B9B47B9AF; S3 Extended Request ID:
 yMah1cwPhqTH267QLPhTKeVPKJB8BO5ndBHzOmWtxLTM6uAvwYYuqieAKyb6qRPTxP1tHXCoR4Y=).
 Message arrived on: error/action, Action: s3, Bucket: us-
east-1-s3-verify-user, Key: \"aaa\". Value of x-amz-id-2:
 yMah1cwPhqTH267QLPhTKeVPKJB8BO5ndBHzOmWtxLTM6uAvwYYuqieAKyb6qRPTxP1tHXCoR4Y="
    }
  ]
}
```

ruleName

The name of the rule that triggered the error action.

topic

The topic in which the original message was received.

cloudwatchTraceId

A unique identity referring to the error logs in CloudWatch.

clientId

The client ID of the message publisher.

base64OriginalPayload

The original message payload Base64-encoded.

failures

failedAction

The name of the action that failed to complete (for example, "S3Action").

failedResource

The name of the resource (for example, the name of an S3 bucket).

errorMessage

The description and explanation of the error.

Error action example

Here is an example of a rule with an added error action. The following rule has an action that writes message data to a DynamoDB table and an error action that writes data to an Amazon S3 bucket:

```
{
    "sql" : "SELECT * FROM ..."
    "actions" : [{
        "dynamoDB" : {
            "table" : "PoorlyConfiguredTable",
```

```
            "hashKeyField" : "AConstantString",
            "hashKeyValue" : "AHashKey"}}
    ],
    "errorAction" : {
        "s3" : {
            "roleArn": "arn:aws:iam::123456789012:role/aws_iot_s3",
            "bucketName" : "message-processing-errors",
            "key" : "${replace(topic(), '/', '-') + '-' + timestamp() + '-' +
 newuuid()}"
        }
    }
}
```

You can use any function or substitution template in an error action's SQL statement including the external functions: `aws_lambda()`, `get_dynamodb()`, `get_registry_data()`, `get_thing_shadow()`, `get_secret()`, `machinelearning_predict()`, and `decode()`. If an error action requires to call an external function, then invoking the error action can result in additional bill for the external function.

For more information about rules and how to specify an error action, see Creating an AWS IoT Rule.

For more information about using CloudWatch to monitor the success or failure of rules, see AWS IoT metrics and dimensions.

Reducing messaging costs with Basic Ingest

You can use Basic Ingest, to securely send device data to the AWS services supported by AWS IoT rule actions, without incurring messaging costs. Basic Ingest optimizes data flow by removing the publish/subscribe message broker from the ingestion path.

Basic Ingest can send messages from your devices or applications. The messages have topic names that start with `$aws/rules/`*`rule_name`* for their first three levels, where *`rule_name`* is the name of the AWS IoT rule that you want to invoke.

You can use an existing rule with Basic Ingest by adding the Basic Ingest prefix (`$aws/rules/`*`rule_name`*) to the message topic that you'd use to invoke the rule. For example, if you have a rule named `BuildingManager` that's invoked by messages with topics like `Buildings/Building5/Floor2/Room201/Lights` (`"sql": "SELECT * FROM 'Buildings/#'"`), you can invoke the same rule with Basic Ingest by sending a message with topic `$aws/rules/BuildingManager/Buildings/Building5/Floor2/Room201/Lights`.

Note

- Your devices and rules can't subscribe to Basic Ingest reserved topics. For example, the AWS IoT Device Defender metric `num-messages-received` metrics is not emitted as it doesn't support subscribing to topics. For more information, see Reserved topics.
- If you need a publish/subscribe broker to distribute messages to multiple subscribers (for example, to deliver messages to other devices and the rules engine), you should continue to use the AWS IoT message broker to handle the message distribution. However, make sure that you publish your messages on topics other than Basic Ingest topics.

Using Basic Ingest

Before you use Basic Ingest, verify that your device or application is using a policy that has publish permissions on `$aws/rules/*`. Or, you can specify permission for individual rules with `$aws/rules/rule_name/*` in the policy. Otherwise, your devices and applications can continue to use their existing connections with AWS IoT Core.

When the message reaches the rules engine, there's no difference in implementation or error handling between rules invoked from Basic Ingest and those invoked through message broker subscriptions.

You can create rules for use with Basic Ingest. Keep in mind the following:

- The initial prefix of a Basic Ingest topic (`$aws/rules/rule_name`) isn't available to the topic(Decimal) function.
- If you define a rule that's invoked only with Basic Ingest, the `FROM` clause is optional in the `sql` field of the `rule` definition. It's still required if the rule is also invoked by other messages that must be sent through the message broker (for example, because those other messages must be distributed to multiple subscribers). For more information, see AWS IoT SQL reference.
- The first three levels of the Basic Ingest topic (`$aws/rules/rule_name`) aren't counted toward the 8-segment length limit or toward the 256-total character limit for a topic. Otherwise, the same restrictions apply as documented in AWS IoT Limits.
- If a message is received with a Basic Ingest topic that specifies an inactive rule or a rule that doesn't exist, an error log is created in an Amazon CloudWatch log to help you with debugging. For more information, see Rules engine log entries. A `RuleNotFound` metric is indicated and you can create alarms on this metric. For more information, see Rule Metrics in Rule metrics.

- You can still publish with QoS 1 on Basic Ingest topics. You receive a PUBACK after the message is successfully delivered to the rules engine. Receiving a PUBACK doesn't mean that your rule actions were completed successfully. You can configure an error action to handle errors when an action is run. For more information, see Error handling (error action).

AWS IoT SQL reference

In AWS IoT, rules are defined using an SQL-like syntax. SQL statements are composed of three types of clauses:

SET

(Optional) Defines variables that you can reuse throughout SQL statements and substitution templates. Assign values to variables using expressions. Reference these variables in SELECT and WHERE clauses, and in action substitution templates.

The SET clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Variables, and Nested object queries.

SELECT

(Required) Extracts information from the payload of an incoming message and performs transformations on the information. The messages to use are identified by the topic filter specified in the FROM clause.

The SELECT clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Substitution templates, Variables, Nested object queries, and Binary payloads.

FROM

The MQTT message topic filter that identifies the messages to extract data from. The rule is activated for each message sent to an MQTT topic that matches the topic filter specified here. Required for rules that are activated by messages that pass through the message broker. Optional for rules that are only activated using the Basic Ingest feature.

WHERE

(Optional) Adds conditional logic that determines whether the actions specified by a rule are carried out.

The WHERE clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Variables, and Nested object queries.

An example SQL statement looks like this:

```
SELECT color AS rgb FROM 'topic/subtopic' WHERE temperature > 50
```

An example MQTT message (also called an incoming payload) looks like this:

```
{
    "color":"red",
    "temperature":100
}
```

If this message is published on the `'topic/subtopic'` topic, the rule is triggered and the SQL statement is evaluated. The SQL statement extracts the value of the `color` property if the `"temperature"` property is greater than 50. The WHERE clause specifies the condition `temperature > 50`. The `AS` keyword renames the `"color"` property to `"rgb"`. The result (also called an *outgoing payload*) looks like this:

```
{
    "rgb":"red"
}
```

This data is then forwarded to the rule's action, which sends the data for more processing. For more information about rule actions, see AWS IoT rule actions.

> **Note**
>
> Comments are not currently supported in AWS IoT SQL syntax.
> Attribute names with spaces in them can't be used as field names in the SQL statement. While the incoming payload can have attribute names with spaces in them, such names can't be used in the SQL statement. They will, however, be passed through to the outgoing payload if you use a wildcard (*) field name specification.

SELECT clause

The AWS IoT SELECT clause is essentially the same as the ANSI SQL SELECT clause, with some minor differences.

The SELECT clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Variables, Nested object queries, and Binary payloads.

You can use the SELECT clause to extract information from incoming MQTT messages. You can also use `SELECT *` to retrieve the entire incoming message payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL statement: SELECT * FROM 'topic/subtopic'
Outgoing payload: {"color":"red", "temperature":50}
```

If the payload is a JSON object, you can reference keys in the object. Your outgoing payload contains the key-value pair. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL statement: SELECT color FROM 'topic/subtopic'
Outgoing payload: {"color":"red"}
```

You can use the AS keyword to rename keys. For example:

```
Incoming payload published on topic 'topic/subtopic':{"color":"red", "temperature":50}
SQL:SELECT color AS my_color FROM 'topic/subtopic'
Outgoing payload: {"my_color":"red"}
```

You can select multiple items by separating them with a comma. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL: SELECT color as my_color, temperature as fahrenheit FROM 'topic/subtopic'
Outgoing payload: {"my_color":"red","fahrenheit":50}
```

You can select multiple items including '*' to add items to the incoming payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL: SELECT *, 15 as speed FROM 'topic/subtopic'
Outgoing payload: {"color":"red", "temperature":50, "speed":15}
```

You can use the `"VALUE"` keyword to produce outgoing payloads that are not JSON objects. With SQL version `2015-10-08`, you can select only one item. With SQL version `2016-03-23` or later, you can also select an array to output as a top-level object.

Example

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL: SELECT VALUE color FROM 'topic/subtopic'
Outgoing payload: "red"
```

You can use '.' syntax to drill into nested JSON objects in the incoming payload. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":
{"red":255,"green":0,"blue":0}, "temperature":50}
SQL: SELECT color.red as red_value FROM 'topic/subtopic'
Outgoing payload: {"red_value":255}
```

For information about how to use JSON object and property names that include reserved characters, such as numbers or the hyphen (minus) character, see JSON extensions

You can use functions (see Functions) to transform the incoming payload. You can use parentheses for grouping. For example:

```
Incoming payload published on topic 'topic/subtopic': {"color":"red", "temperature":50}
SQL: SELECT (temperature - 32) * 5 / 9 AS celsius, upper(color) as my_color FROM
 'topic/subtopic'
Outgoing payload: {"celsius":10,"my_color":"RED"}
```

FROM clause

The FROM clause subscribes your rule to a topic or topic filter. Enclose the topic or topic filter in single quotes ('). The rule is triggered for each message sent to an MQTT topic that matches the topic filter specified here. You can subscribe to a group of similar topics using a topic filter.

Example:

Incoming payload published on topic `'topic/subtopic'`: `{temperature: 50}`

Incoming payload published on topic `'topic/subtopic-2'`: `{temperature: 50}`

SQL: `"SELECT temperature AS t FROM 'topic/subtopic'"`.

The rule is subscribed to `'topic/subtopic'`, so the incoming payload is passed to the rule. The outgoing payload, passed to the rule actions, is: `{t: 50}`. The rule is not subscribed to `'topic/subtopic-2'`, so the rule is not triggered for the message published on `'topic/subtopic-2'`.

 # Wildcard Example:

You can use the '#' (multi-level) wildcard character to match one or more particular path elements:

Incoming payload published on topic `'topic/subtopic'`: `{temperature: 50}`.

Incoming payload published on topic `'topic/subtopic-2'`: `{temperature: 60}`.

Incoming payload published on topic `'topic/subtopic-3/details'`: `{temperature: 70}`.

Incoming payload published on topic `'topic-2/subtopic-x'`: `{temperature: 80}`.

SQL: `"SELECT temperature AS t FROM 'topic/#'"`.

The rule is subscribed to any topic that begins with `'topic'`, so it's executed three times, sending outgoing payloads of `{t: 50}` (for topic/subtopic), `{t: 60}` (for topic/subtopic-2), and `{t: 70}` (for topic/subtopic-3/details) to its actions. It's not subscribed to `'topic-2/subtopic-x'`, so the rule isn't triggered for the `{temperature: 80}` message.

+ Wildcard Example:

You can use the '+' (single-level) wildcard character to match any one particular path element:

Incoming payload published on topic `'topic/subtopic'`: `{temperature: 50}`.

Incoming payload published on topic `'topic/subtopic-2'`: `{temperature: 60}`.

Incoming payload published on topic `'topic/subtopic-3/details'`: `{temperature: 70}`.

Incoming payload published on topic `'topic-2/subtopic-x'`: `{temperature: 80}`.

SQL: `"SELECT temperature AS t FROM 'topic/+'"`.

The rule is subscribed to all topics with two path elements where the first element is `'topic'`. The rule is executed for the messages sent to `'topic/subtopic'` and `'topic/subtopic-2'`, but not `'topic/subtopic-3/details'` (it has more levels than the topic filter) or `'topic-2/subtopic-x'` (it doesn't start with `topic`).

SET clause

Use the SET clause to define variables that store expression results. You can reuse these variables in SELECT and WHERE clauses, and in substitution templates. This helps you avoid duplicating complex expressions and reduce the number of function calls in your SQL statement.

The SET clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Variables, and Nested object queries.

SET clause syntax

The SET clause must appear before the SELECT clause in your SQL statement. Use the following syntax:

```
SET @variable_name = expression [, @variable_name2 = expression2]
```

Syntax rules:

- Start variable names with `@`
- Variable names can contain letters, numbers, and underscores
- Variable names can be up to 64 characters long
- Multiple variables can be set in a single SET clause, separated by commas
- Each variable can only be assigned once (variables are immutable)
- The SET keyword can only be used once per SQL statement

Using variables

After you define variables, you can use them in:

- SELECT clauses
- WHERE clauses
- Other SET variable assignments
- Action substitution templates
- Error action substitution templates
- Nested SELECT queries
- Function parameters (certain parameters such as roleArn parameters and parameters that switch the mode of a function similar to `transform("enrichArray", attributes, values)` do not support variables)

Variables are referenced using the same `@variable_name` syntax used in the SET clause. You can also use JSON extension syntax to access properties of variables that contain objects, such as `@variable_name.property`.

SET clause examples

Basic variable usage

The following example shows a payload published on topic `device/data`: `{"temp_fahrenheit": 75, "humidity": 60}`

SQL statement:

```
SET @temp_celsius = (temp_fahrenheit - 32) * 5 / 9
SELECT @temp_celsius AS celsius, humidity FROM 'device/data'
```

Outgoing payload: `{"celsius": 23.89, "humidity": 60}`

Access members in embedded JSON objects

The following example shows a payload published on topic `device/data`: `{"device1": {"deviceId":"weather_sensor", "deviceData": {"sensors": {"temp_fahrenheit": 75, "humidity": 60}, "location": [47.606,-122.332]}}}`

SQL statement:

```
SET @device_sensor_data = device1.deviceData.sensors
SELECT @device_sensor_data.temp_fahrenheit AS temp_fahrenheit,
 @device_sensor_data.humidity as humidity, device1.deviceId as deviceId FROM 'device/
data'
```

Outgoing payload: `{"temp_fahrenheit":75,"humidity":60,"deviceId":"weather_sensor"}`

for more information on how to work with JSON extensions, reference JSON extensions

Avoiding duplicate function calls

SET variables help avoid duplicating complex decode operations:

```
SET @decoded_data = decode(encode(*, 'base64'), 'proto', 'schema', 'schema.desc',
 'message.proto', 'Message')
SELECT @decoded_data.sensor_id, @decoded_data.reading FROM 'device/protobuf'
WHERE @decoded_data.reading > 100
```

Without SET variables, you would need to repeat the decode function three times, which exceeds the function call limits.

Multiple variables

You can define multiple variables in a single SET clause by separating them with commas:

```
SET @user_data = get_user_properties(device_id), @threshold = 50
SELECT @user_data.name, temp_fahrenheit FROM 'sensors/+'
WHERE temp_fahrenheit > @threshold AND @user_data.active = true
```

Using variables in substitution templates

Variables can also be used in action substitution templates, allowing you to reuse computed values across both the SQL statement and rule actions.

SQL statement:

```
SET @temp_celsius = (temp_fahrenheit - 32) * 5 / 9
SELECT @temp_celsius AS celsius, humidity FROM 'device/data'
```

Action configuration:

```
{
  "s3": {
    "roleArn": "arn:aws:iam::123456789012:role/testRuleRole",
    "bucketName": "bucket",
    "key": "temperature-data/${device_id}/temp-${@temp_celsius}C.json"
  }
}
```

In this example, the SET variable `@temp_celsius` is used in a substitution template to construct the key field of the S3 action.

Non-JSON payload usage

SET variables does not support non-JSON payloads directly, so the payload must be encoded or decoded first:

```
SET @encoded_payload = encode(*, 'base64')
SELECT @encoded_payload AS raw_data FROM 'device/binary'
```

for more information on how to work with non-JSON payloads, reference Working with binary payloads

SET clause limits

The following limits apply to SET variables:

- Maximum of 10 unique variables per SQL statement
- Maximum variable value size of 128 KiB (minified UTF-8 JSON string)
- Maximum total value size of 128 KiB for all variables
- Variable names limited to 64 characters
- Variables can accept JSON payloads directly as is (non-JSON payloads must first be encoded/decoded)

WHERE clause

The WHERE clause determines if the actions specified by a rule are carried out. If the WHERE clause evaluates to true, the rule actions are performed. Otherwise, the rule actions are not performed.

The WHERE clause supports Data types, Operators, Functions, Literals, Case statements, JSON extensions, Variables, and Nested object queries.

Example:

Incoming payload published on `topic/subtopic`: `{"color":"red", "temperature":40}`.

SQL: `SELECT color AS my_color FROM 'topic/subtopic' WHERE temperature > 50 AND color <> 'red'`.

In this case, the rule will be triggered, but the actions specified by the rule will not be performed. There will be no outgoing payload.

You can use functions and operators in the WHERE clause. However, you cannot reference any aliases created with the AS keyword in the SELECT. The WHERE clause is evaluated first, to determine if SELECT is evaluated.

Example with non-JSON payload:

Incoming non-JSON payload published on `topic/subtopic`: `80`

SQL: `` `SELECT decode(encode(*, 'base64'), 'base64') AS value FROM 'topic/subtopic' WHERE decode(encode(*, 'base64'), 'base64') > 50 ``

In this case, the rule will be triggered, and the actions specified by the rule will be performed. The outgoing payload will be transformed by the SELECT clause as a JSON payload `{"value":80}`.

Data types

The AWS IoT rules engine supports all JSON data types.

Supported data types

Type	Meaning
`Int`	A discrete `Int`. 34 digits maximum.
`Decimal`	A `Decimal` with a precision of 34 digits, with a minimum non-zero magnitude of 1E-999 and a maximum magnitude 9.999...E999. **Note** Some functions return `Decimal` values with double precision rather than 34-digit precision. With SQL V2 (2016-03-23), numeric values that are whole numbers, such as `10.0`, are processed as an `Int` value (`10`) instead of the expected `Decimal` value (`10.0`). To reliably process whole number numeric values as `Decimal` values, use SQL V1 (2015-10-08) for the rule query statement.
`Boolean`	`True` or `False`.
`String`	A UTF-8 string.

Type	Meaning
`Array`	A series of values that don't have to have the same type.
`Object`	A JSON value consisting of a key and a value. Keys must be strings. Values can be any type.
`Null`	`Null` as defined by JSON. It's an actual value that represents the absence of a value. You can explicitly create a `Null` value by using the `Null` keyword in your SQL statement. For example: `"SELECT NULL AS n FROM 'topic/subtopic'"`

`Undefined`	Not a value. This isn't explicitly represent able in JSON except by omitting the value. For example, in the object `{"foo": null}`, the key "foo" returns NULL, but the key "bar" returns `Undefined` . Internally, the SQL language treats `Undefined` as a value, but it isn't representable in JSON, so when serialized to JSON, the results are `Undefined` .
	`{"foo":null, "bar":undefined}`
	is serialized to JSON as:
	`{"foo":null}`
	Similarly, `Undefined` is converted to an empty string when serialized by itself. Functions called with invalid arguments (for example, wrong types, wrong number of arguments, and so on) return `Undefined` .

Conversions

The following table lists the results when a value of one type is converted to another type (when a value of the incorrect type is given to a function). For example, if the absolute value function "abs" (which expects an `Int` or `Decimal`) is given a `String`, it attempts to convert the `String` to a `Decimal`, following these rules. In this case, 'abs("-5.123")' is treated as 'abs(-5.123)'.

> **Note**
>
> There are no attempted conversions to `Array`, `Object`, `Null`, or `Undefined`.

To decimal

Argument type	Result
`Int`	A `Decimal` with no decimal point.
`Decimal`	The source value.
`Boolean`	`Undefined` . (You can explicitly use the cast function to transform true = 1.0, false = 0.0.)
`String`	The SQL engine tries to parse the string as a `Decimal`. AWS IoT attempts to parse strings matching the regular expression:`^-?\d+(\.\d+)?((?i)E-?\d+)?$` . "0", "-1.2", "5E-12" are all examples of strings that are converted automatically to `Decimal`s.
Array	`Undefined` .
Object	`Undefined` .
Null	`Null`.
Undefined	`Undefined` .

To int

Argument type	Result
`Int`	The source value.
`Decimal`	The source value rounded to the nearest `Int`.
`Boolean`	`Undefined` . (You can explicitly use the cast function to transform true = 1.0, false = 0.0.)
`String`	The SQL engine tries to parse the string as a `Decimal`. AWS IoT attempts to parse strings matching the regular expression:`^-?\d+(\.\d+)?((?i)E-?\d+)?$` . "0", "-1.2", "5E-12" are all examples of strings that are converted automatically to `Decimals`. AWS IoT attempts to convert the `String` to a `Decimal`, and then truncates the decimal places of that `Decimal` to make an `Int`.
Array	`Undefined` .
Object	`Undefined` .
Null	`Null`.
Undefined	`Undefined` .

To Boolean

Argument type	Result
`Int`	`Undefined` . (You can explicitly use the `cast` function to transform 0 = False, any_nonzero_value = True.)

Argument type	Result
`Decimal`	`Undefined` . (You can explicitly use the cast function to transform 0 = False, any_nonze ro_value = True.)
`Boolean`	The original value.
`String`	"true"=True and "false"=False (case insensiti ve). Other string values are `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

To string

Argument type	Result
`Int`	A string representation of the `Int` in standard notation.
`Decimal`	A string representing the `Decimal` value, possibly in scientific notation.
`Boolean`	"true" or "false". All lowercase.
`String`	The original value.
Array	The `Array` serialized to JSON. The resultant string is a comma-separated list, enclosed in square brackets. A `String` is quoted. A `Decimal`, `Int`, `Boolean`, and `Null` is not.
Object	The object serialized to JSON. The resultant string is a comma-separated list of key-value

Argument type	Result
	pairs and begins and ends with curly braces. A `String` is quoted. A `Decimal`, `Int`, `Boolean`, and `Null` is not.
Null	`Undefined` .
Undefined	Undefined.

Operators

The following operators can be used in SELECT and WHERE clauses.

AND operator

Returns a `Boolean` result. Performs a logical AND operation. Returns true if left and right operands are true. Otherwise, returns false. `Boolean` operands or case insensitive "true" or "false" string operands are required.

Syntax: `expression AND expression`.

AND operator

Left operand	Right operand	Output
`Boolean`	`Boolean`	`Boolean`. True if both operands are true. Otherwise, false.
`String/Boolean`	`String/Boolean`	If all strings are "true" or "false" (case insensitive), they are converted to `Boolean` and processed normally as `boolean AND boolean`.
Other value	Other value	`Undefined` .

OR operator

Returns a `Boolean` result. Performs a logical OR operation. Returns true if either the left or the right operands are true. Otherwise, returns false. `Boolean` operands or case insensitive "true" or "false" string operands are required.

Syntax: `expression OR expression`.

OR operator

Left operand	Right operand	Output
`Boolean`	`Boolean`	`Boolean`. True if either operand is true. Otherwise, false.
`String/Boolean`	`String/Boolean`	If all strings are "true" or "false" (case insensitive), they are converted to Booleans and processed normally as `boolean OR boolean`.
Other value	Other value	`Undefined` .

NOT operator

Returns a `Boolean` result. Performs a logical NOT operation. Returns true if the operand is false. Otherwise, returns true. A `Boolean` operand or case insensitive "true" or "false" string operand is required.

Syntax: `NOT expression`.

NOT operator

Operand	Output
`Boolean`	`Boolean`. True if operand is false. Otherwise, true.
`String`	If string is "true" or "false" (case insensitive), it is converted to the corresponding Boolean value, and the opposite value is returned.

Operand	Output
Other value	`Undefined` .

IN operator

Returns a `Boolean` result. You can use the IN operator in a WHERE clause to check if a value matches any value in an array. It returns true if the match is found, and false otherwise.

Syntax: *`expression`* `IN` *`expression`*.

IN operator

Left operand	Right operand	Output
`Int/Decimal/String/`	`Array`	True if the `Integer/Decimal/String/Array/Object` element is found in the array. Otherwise, false.

Example:

```
SQL: "select * from 'a/b' where 3 in arr"

JSON: {"arr":[1, 2, 3, "three", 5.7, null]}
```

In this example, the condition clause `where 3 in arr` will evaluate to true because 3 is present in the array named `arr`. Hence in the SQL statement, `select * from 'a/b'` will execute. This example also shows that the array can be heterogeneous.

EXISTS operator

Returns a `Boolean` result. You can use the EXISTS operator in a conditional clause to test for the existence of elements in a subquery. It returns true if the subquery returns one or more elements and false if the subquery returns no elements.

Syntax: *`expression`*.

Example:

```
SQL: "select * from 'a/b' where exists (select * from arr as a where a = 3)"

JSON: {"arr":[1, 2, 3]}
```

In this example, the condition clause `where exists (select * from arr as a where a = 3)` will evaluate to true because 3 is present in the array named `arr`. Hence in the SQL statement, `select * from 'a/b'` will execute.

Example:

```
SQL: select * from 'a/b' where exists (select * from e as e where foo = 2)

JSON: {"foo":4,"bar":5,"e":[{"foo":1},{"foo":2}]}
```

In this example, the condition clause `where exists (select * from e as e where foo = 2)` will evaluate to true because the array `e` within the JSON object contains the object `{"foo":2}`. Hence in the SQL statement, `select * from 'a/b'` will execute.

> operator

Returns a `Boolean` result. Returns true if the left operand is greater than the right operand. Both operands are converted to a `Decimal`, and then compared.

Syntax: *`expression`* `>` *`expression`*.

> operator

Left operand	Right operand	Output
`Int/Decimal`	`Int/Decimal`	`Boolean`. True if the left operand is greater than the right operand. Otherwise, false.
`String/Int/Dec:`	`String/Int/Dec:`	If all strings can be converted to `Decimal`, then `Boolean`. Returns true if the left operand is greater than the right operand. Otherwise, false.
Other value	`Undefined`.	`Undefined`.

>= operator

Returns a `Boolean` result. Returns true if the left operand is greater than or equal to the right operand. Both operands are converted to a `Decimal`, and then compared.

Syntax: `expression >= expression`.

>= operator

Left operand	Right operand	Output
`Int/Decimal`	`Int/Decimal`	`Boolean`. True if the left operand is greater than or equal to the right operand. Otherwise, false.
`String/Int/Dec`	`String/Int/Dec`	If all strings can be converted to `Decimal`, then `Boolean`. Returns true if the left operand is greater than or equal to the right operand. Otherwise, false.
Other value	`Undefined` .	`Undefined` .

< operator

Returns a `Boolean` result. Returns true if the left operand is less than the right operand. Both operands are converted to a `Decimal`, and then compared.

Syntax: `expression < expression`.

< operator

Left operand	Right operand	Output
`Int/Decimal`	`Int/Decimal`	`Boolean`. True if the left operand is less than the right operand. Otherwise, false.
`String/Int/Dec`	`String/Int/Dec`	If all strings can be converted to `Decimal`, then `Boolean`. Returns true if the left operand is less than the right operand. Otherwise, false.
Other value	`Undefined`	`Undefined`

<= operator

Returns a `Boolean` result. Returns true if the left operand is less than or equal to the right operand. Both operands are converted to a `Decimal`, and then compared.

Syntax: *`expression`* `<=` *`expression`*.

<= operator

Left operand	Right operand	Output
`Int/Decimal`	`Int/Decimal`	`Boolean`. True if the left operand is less than or equal to the right operand. Otherwise, false.
`String/Int/Dec`	`String/Int/Dec`	If all strings can be converted to `Decimal`, then `Boolean`. Returns true if the left operand is less than or equal to the right operand. Otherwise, false.
Other value	`Undefined`	`Undefined`

<> operator

Returns a `Boolean` result. Returns true if both left and right operands are not equal. Otherwise, returns false.

Syntax: *`expression`* `<>` *`expression`*.

<> operator

Left operand	Right operand	Output
`Int`	`Int`	True if left operand is not equal to right operand. Otherwise, false.
`Decimal`	`Decimal`	True if left operand is not equal to right operand. Otherwise, false.`Int` is converted to `Decimal` before being compared.
`String`	`String`	True if left operand is not equal to right operand. Otherwise, false.

Left operand	Right operand	Output
Array	Array	True if the items in each operand are not equal and not in the same order. Otherwise, false
Object	Object	True if the keys and values of each operand are not equal. Otherwise, false. The order of keys/values is unimportant.
Null	Null	False.
Any value	`Undefined`	Undefined.
`Undefined`	Any value	Undefined.
Mismatched type	Mismatched type	True.

= operator

Returns a `Boolean` result. Returns true if both left and right operands are equal. Otherwise, returns false.

Syntax: `expression = expression`.

= operator

Left operand	Right operand	Output
`Int`	`Int`	True if left operand is equal to right operand. Otherwise , false.
`Decimal`	`Decimal`	True if left operand is equal to right operand. Otherwise , false.`Int` is converted to `Decimal` before being compared.
`String`	`String`	True if left operand is equal to right operand. Otherwise , false.

Left operand	Right operand	Output
Array	Array	True if the items in each operand are equal and in the same order. Otherwise, false.
Object	Object	True if the keys and values of each operand are equal. Otherwise, false. The order of keys/values is unimporta nt.
Any value	`Undefined`	`Undefined` .
`Undefined`	Any value	`Undefined` .
Mismatched type	Mismatched type	False.

+ operator

The "+" is an overloaded operator. It can be used for string concatenation or addition.

Syntax: `expression + expression`.

+ operator

Left operand	Right operand	Output
`String`	Any value	Converts the right operand to a string and concatenates it to the end of the left operand.
Any value	`String`	Converts the left operand to a string and concatena tes the right operand to the end of the converted left operand.
`Int`	`Int`	`Int` value. Adds operands together.
`Int/Decimal`	`Int/Decimal`	`Decimal` value. Adds operands together.
Other value	Other value	`Undefined` .

- operator

Subtracts the right operand from the left operand.

Syntax: `expression - expression`.

- operator

Left operand	Right operand	Output
`Int`	`Int`	`Int` value. Subtracts right operand from left operand.
`Int/Decimal`	`Int/Decimal`	`Decimal` value. Subtracts right operand from left operand.
`String/Int/Dec:`	`String/Int/Dec:`	If all strings convert to decimals correctly, a `Decimal` value is returned. Subtracts right operand from left operand. Otherwise, returns `Undefined` .
Other value	Other value	`Undefined` .
Other value	Other value	`Undefined` .

* operator

Multiplies the left operand by the right operand.

Syntax: `expression * expression`.

*** operator**

Left operand	Right operand	Output
`Int`	`Int`	`Int` value. Multiplies the left operand by the right operand.
`Int/Decimal`	`Int/Decimal`	`Decimal` value. Multiplies the left operand by the right operand.

Left operand	Right operand	Output
`String/Int/Dec`	`String/Int/Dec`	If all strings convert to decimals correctly, a `Decimal` value is returned. Multiplies the left operand by the right operand. Otherwise, returns `Undefined` .
Other value	Other value	`Undefined` .

/ operator

Divides the left operand by the right operand.

Syntax: *`expression`* `/` *`expression`*.

/ operator

Left operand	Right operand	Output
`Int`	`Int`	`Int` value. Divides the left operand by the right operand.
`Int/Decimal`	`Int/Decimal`	`Decimal` value. Divides the left operand by the right operand.
`String/Int/Dec`	`String/Int/Dec`	If all strings convert to decimals correctly, a `Decimal` value is returned. Divides the left operand by the right operand. Otherwise, returns `Undefined` .
Other value	Other value	`Undefined` .

% operator

Returns the remainder from dividing the left operand by the right operand.

Syntax: *`expression`* `%` *`expression`*.

% operator

Left operand	Right operand	Output
Int	Int	Int value. Returns the remainder from dividing the left operand by the right operand.
String/Int/Dec:	String/Int/Dec:	If all strings convert to decimals correctly, a Decimal value is returned. Returns the remainder from dividing the left operand by the right operand. Otherwise, Undefined .
Other value	Other value	Undefined .

Functions

You can use the following built-in functions in the SELECT or WHERE clauses of your SQL expressions.

The following external functions are billed equivalent to that of a rule action: aws_lambda, get_dynamodb(), get_registry_data(), and get_thing_shadow(). You also get billed for the decode() function only when you are decoding a Protobuf message to JSON. For more details, refer to the AWS IoT Core pricing page.

abs(Decimal)

Returns the absolute value of a number. Supported by SQL version 2015-10-08 and later.

Example: abs(-5) returns 5.

Argument type	Result
Int	Int, the absolute value of the argument.
Decimal	Decimal, the absolute value of the argument.
Boolean	Undefined .

Argument type	Result
`String`	`Decimal`. The result is the absolute value of the argument. If the string cannot be converted, the result is `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

accountid()

Returns the ID of the account that owns this rule as a `String`. Supported by SQL version 2015-10-08 and later.

Example:

`accountid()` = "123456789012"

acos(Decimal)

Returns the inverse cosine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `acos(0)` = 1.5707963267948966

Argument type	Result
`Int`	`Decimal` (with double precision), the inverse cosine of the argument. Imaginary results are returned as `Undefined`.
`Decimal`	`Decimal` (with double precision), the inverse cosine of the argument. Imaginary results are returned as `Undefined`.

Argument type	Result
`Boolean`	`Undefined` .
`String`	`Decimal`, the inverse cosine of the argument. If the string cannot be converted, the result is `Undefined` . Imaginary results are returned as `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

asin(Decimal)

Returns the inverse sine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `asin(0)` = 0.0

Argument type	Result
`Int`	`Decimal` (with double precision), the inverse sine of the argument. Imaginary results are returned as `Undefined` .
`Decimal`	`Decimal` (with double precision), the inverse sine of the argument. Imaginary results are returned as `Undefined` .
`Boolean`	`Undefined` .
`String`	`Decimal` (with double precision), the inverse sine of the argument. If the

Argument type	Result
	string cannot be converted, the result is `Undefined` . Imaginary results are returned as `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

atan(Decimal)

Returns the inverse tangent of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `atan(0)` = 0.0

Argument type	Result
`Int`	`Decimal` (with double precision), the inverse tangent of the argument. Imaginary results are returned as `Undefined` .
`Decimal`	`Decimal` (with double precision), the inverse tangent of the argument. Imaginary results are returned as `Undefined` .
`Boolean`	`Undefined` .
`String`	`Decimal`, the inverse tangent of the argument. If the string cannot be converted, the result is `Undefined`

Argument type	Result
	. Imaginary results are returned as `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

atan2(Decimal, Decimal)

Returns the angle, in radians, between the positive x-axis and the (x, y) point defined in the two arguments. The angle is positive for counter-clockwise angles (upper half-plane, y > 0), and negative for clockwise angles (lower half-plane, y < 0). `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `atan2(1, 0)` = 1.5707963267948966

Argument type	Argument type	Result
`Int/Decimal`	`Int/Decimal`	`Decimal` (with double precisio angle between the x-axis and t (x,y) point.
`Int/Decimal/String`	`Int/Decimal/String`	`Decimal`, the inverse tangent c described. If a string cannot be the result is `Undefined` .
Other value	Other value	`Undefined` .

aws_lambda(functionArn, inputJson)

Calls the specified Lambda function passing `inputJson` to the Lambda function and returns the JSON generated by the Lambda function.

Arguments

Argument	Description
`functionArn`	The ARN of the Lambda function to call. The Lambda function must return JSON data.
`inputJson`	The JSON input passed to the Lambda function. To pass nested object queries and literals, you must use SQL version 2016-03-23.

You must grant AWS IoT `lambda:InvokeFunction` permissions to invoke the specified Lambda function. The following example shows how to grant the `lambda:InvokeFunction` permission using the AWS CLI:

```
aws lambda add-permission --function-name "function_name"
--region "region"
--principal iot.amazonaws.com
--source-arn arn:aws:iot:us-east-1:account_id:rule/rule_name
--source-account "account_id"
--statement-id "unique_id"
--action "lambda:InvokeFunction"
```

The following are the arguments for the **add-permission** command:

--function-name

Name of the Lambda function. You add a new permission to update the function's resource policy.

--region

The AWS Region of your account.

--principal

The principal who is getting the permission. This should be `iot.amazonaws.com` to allow AWS IoT permission to call a Lambda function.

--source-arn

The ARN of the rule. You can use the **get-topic-rule** AWS CLI command to get the ARN of a rule.

--source-account

The AWS account where the rule is defined.

--statement-id

A unique statement identifier.

--action

The Lambda action that you want to allow in this statement. To allow AWS IoT to invoke a Lambda function, specify `lambda:InvokeFunction`.

> **Important**
>
> If you add a permission for an AWS IoT principal without providing the `source-arn` or `source-account`, any AWS account that creates a rule with your Lambda action can trigger rules to invoke your Lambda function from AWS IoT. For more information, see Lambda Permission Model.

Given a JSON message payload like:

```
{
    "attribute1": 21,
    "attribute2": "value"
}
```

The `aws_lambda` function can be used to call Lambda function as follows.

```
SELECT
aws_lambda("arn:aws:lambda:us-east-1:account_id:function:lambda_function",
 {"payload":attribute1}) as output FROM 'topic-filter'
```

If you want to pass the full MQTT message payload, you can specify the JSON payload using '*', such as the following example.

```
SELECT
aws_lambda("arn:aws:lambda:us-east-1:account_id:function:lambda_function", *) as output
 FROM 'topic-filter'
```

`payload.inner.element` selects data from messages published on topic 'topic/subtopic'.

`some.value` selects data from the output that's generated by the Lambda function.

Note

The rules engine limits the execution duration of Lambda functions. Lambda function calls from rules should be completed within 2000 milliseconds.

bitand(Int, Int)

Performs a bitwise AND on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-08 and later.

Example: `bitand(13, 5)` = 5

Argument type	Argument type	Result
`Int`	`Int`	`Int`, a bitwise AND of the two a
`Int/Decimal`	`Int/Decimal`	`Int`, a bitwise AND of the two a . All non-`Int` numbers are roun to the nearest `Int`. If any of the cannot be converted to an `Int` is `Undefined` .
`Int/Decimal/String`	`Int/Decimal/String`	`Int`, a bitwise AND of the two a . All strings are converted to de and are rounded down to the n `Int`. If the conversion fails, the `Undefined` .
Other value	Other value	`Undefined` .

bitor(Int, Int)

Performs a bitwise OR of the bit representations of the two arguments. Supported by SQL version 2015-10-08 and later.

Example: `bitor(8, 5)` = 13

Argument type	Argument type	Result
`Int`	`Int`	`Int`, the bitwise OR of the two
`Int/Decimal`	`Int/Decimal`	`Int`, the bitwise OR of the two All non-`Int` numbers are round the nearest `Int`. If the conversi result is `Undefined` .
`Int/Decimal/String`	`Int/Decimal/String`	`Int`, the bitwise OR on the two . All strings are converted to de rounded down to the nearest I conversion fails, the result is Ur
Other value	Other value	`Undefined` .

bitxor(Int, Int)

Performs a bitwise XOR on the bit representations of the two `Int`(-converted) arguments. Supported by SQL version 2015-10-08 and later.

Example:`bitor(13, 5)` = 8

Argument type	Argument type	Result
`Int`	`Int`	`Int`, a bitwise XOR on the two
`Int/Decimal`	`Int/Decimal`	`Int`, a bitwise XOR on the two . Non-`Int` numbers are rounde the nearest `Int`.
`Int/Decimal/String`	`Int/Decimal/String`	`Int`, a bitwise XOR on the two . strings are converted to decim rounded down to the nearest I conversion fails, the result is Ur

Argument type	Argument type	Result
Other value	Other value	`Undefined` .

bitnot(Int)

Performs a bitwise NOT on the bit representations of the `Int`(-converted) argument. Supported by SQL version 2015-10-08 and later.

Example: `bitnot(13)` = 2

Argument type	Result
`Int`	`Int`, a bitwise NOT of the argument.
`Decimal`	`Int`, a bitwise NOT of the argument. The `Decimal` value is rounded down to the nearest `Int`.
`String`	`Int`, a bitwise NOT of the argument. Strings are converted to decimals and rounded down to the nearest `Int`. If any conversion fails, the result is `Undefined` .
Other value	Other value.

cast()

Converts a value from one data type to another. Cast behaves mostly like the standard conversions, with the addition of the ability to cast numbers to or from Booleans. If AWS IoT cannot determine how to cast one type to another, the result is `Undefined`. Supported by SQL version 2015-10-08 and later. Format: cast(*`value`* as *`type`*).

Example:

`cast(true as Int)` = 1

The following keywords might appear after "as" when calling `cast`:

For SQL version 2015-10-08 and 2016-03-23

Keyword	Result
`String`	Casts value to `String`.
Nvarchar	Casts value to `String`.
Text	Casts value to `String`.
Ntext	Casts value to `String`.
varchar	Casts value to `String`.
`Int`	Casts value to `Int`.
Integer	Casts value to `Int`.
Double	Casts value to `Decimal` (with double precision).

Additionally, for SQL version 2016-03-23

Keyword	Result
`Decimal`	Casts value to `Decimal`.
Bool	Casts value to `Boolean`.
`Boolean`	Casts value to `Boolean`.

Casting rules:

Cast to decimal

Argument type	Result
`Int`	A `Decimal` with no decimal point.
`Decimal`	The source value.

Argument type	Result
	Note With SQL V2 (2016-03-23), numeric values that are whole numbers, such as `10.0`, return an `Int` value (`10`) instead of the expected `Decimal` value (`10.0`). To reliably cast whole number numeric values as `Decimal` values, use SQL V1 (2015-10-08) for the rule query statement.
`Boolean`	true = 1.0, false = 0.0.
`String`	Tries to parse the string as a `Decimal`. AWS IoT attempts to parse strings matching the regex: ^-?\d+(\.\d+)?((?i)E-?\d+)?$. "0", "-1.2", "5E-12" are all examples of strings that are converted automatically to decimals.
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

Cast to int

Argument type	Result
`Int`	The source value.

Argument type	Result
`Decimal`	The source value, rounded down to the nearest `Int`.
`Boolean`	true = 1.0, false = 0.0.
`String`	Tries to parse the string as a `Decimal`. AWS IoT attempts to parse strings matching the regex: ^-?\d+(\.\d+)?((?i)E-?\d+)?$. "0", "-1.2", "5E-12" are all examples of strings that are converted automatically to decimals. AWS IoT attempts to convert the string to a `Decimal` and round down to the nearest `Int`.
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

Cast to `Boolean`

Argument type	Result
`Int`	0 = False, any_nonzero_value = True.
`Decimal`	0 = False, any_nonzero_value = True.
`Boolean`	The source value.
`String`	"true" = True and "false" = False (case insensitive). Other string values = `Undefined` .
Array	`Undefined` .

Argument type	Result
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

Cast to string

Argument type	Result
`Int`	A string representation of the `Int`, in standard notation.
`Decimal`	A string representing the `Decimal` value, possibly in scientific notation.
`Boolean`	"true" or "false", all lowercase.
`String`	The source value.
Array	The array serialized to JSON. The result string is a comma-separated list enclosed in square brackets. `String` is quoted. `Decimal`, `Int`, and `Boolean` are not.
Object	The object serialized to JSON. The JSON string is a comma-separated list of key-value pairs and begins and ends with curly braces. `String` is quoted. `Decimal`, `Int`, `Boolean`, and `Null` are not.
Null	`Undefined` .
Undefined	`Undefined` .

ceil(Decimal)

Rounds the given `Decimal` up to the nearest `Int`. Supported by SQL version 2015-10-08 and later.

Examples:

`ceil(1.2)` = 2

`ceil(-1.2)` = -1

Argument type	Result
`Int`	`Int`, the argument value.
`Decimal`	`Int`, the `Decimal` value rounded up to the nearest `Int`.
`String`	`Int`. The string is converted to `Decimal` and rounded up to the nearest `Int`. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Other value	`Undefined` .

chr(String)

Returns the ASCII character that corresponds to the given `Int` argument. Supported by SQL version 2015-10-08 and later.

Examples:

`chr(65)` = "A".

`chr(49)` = "1".

Argument type	Result
`Int`	The character corresponding to the specified ASCII value. If the argument

Argument type	Result
	is not a valid ASCII value, the result is `Undefined`.
`Decimal`	The character corresponding to the specified ASCII value. The `Decimal` argument is rounded down to the nearest `Int`. If the argument is not a valid ASCII value, the result is `Undefined`.
`Boolean`	`Undefined`.
`String`	If the `String` can be converted to a `Decimal`, it is rounded down to the nearest `Int`. If the argument is not a valid ASCII value, the result is `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Other value	`Undefined`.

clientid()

Returns the ID of the MQTT client sending the message, or `n/a` if the message wasn't sent over MQTT. Supported by SQL version 2015-10-08 and later.

Example:

`clientid()` = "123456789012"

concat()

Concatenates arrays or strings. This function accepts any number of arguments and returns a `String` or an `Array`. Supported by SQL version 2015-10-08 and later.

Examples:

`concat()` = `Undefined`.

`concat(1)` = "1".

`concat([1, 2, 3], 4)` = [1, 2, 3, 4].

`concat([1, 2, 3], "hello")` = [1, 2, 3, "hello"]

`concat("con", "cat")` = "concat"

`concat(1, "hello")` = "1hello"

`concat("he","is","man")` = "heisman"

`concat([1, 2, 3], "hello", [4, 5, 6])` = [1, 2, 3, "hello", 4, 5, 6]

Number of arguments	Result
0	`Undefined`.
1	The argument is returned unmodified.
2+	If any argument is an `Array`, the result is a single array containing all of the arguments. If no arguments are arrays, and at least one argument is a `String`, the result is the concatenation of the `String` representations of all the arguments. Arguments are converted to strings using the standard conversions previously listed.

cos(Decimal)

Returns the cosine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example:

`cos(0)` = 1.

Argument type	Result
`Int`	`Decimal` (with double precision), the cosine of the argument. Imaginary results are returned as `Undefined`.
`Decimal`	`Decimal` (with double precision), the cosine of the argument. Imaginary results are returned as `Undefined`.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the cosine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined`. Imaginary results are returned as `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

cosh(Decimal)

Returns the hyperbolic cosine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `cosh(2.3)` = 5.037220649268761.

Argument type	Result
`Int`	`Decimal` (with double precision), the hyperbolic cosine of the argument.

Argument type	Result
	Imaginary results are returned as `Undefined`.
`Decimal`	`Decimal` (with double precision), the hyperbolic cosine of the argument. Imaginary results are returned as `Undefined`.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the hyperbolic cosine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined`. Imaginary results are returned as `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

decode(value, decodingScheme)

Use the `decode` function to decode an encoded value. If the decoded string is a JSON document, an addressable object is returned. Otherwise, the decoded string is returned as a string. The function returns NULL if the string cannot be decoded. This function supports decoding base64-encoded strings and Protocol Buffer (protobuf) message format.

Supported by SQL version 2016-03-23 and later.

value

A string value or any of the valid expressions, as defined in AWS IoT SQL reference, that return a string.

decodingScheme

A literal string representing the scheme used to decode the value. Currently, only 'base64' and 'proto' are supported.

Decoding base64-encoded strings

In this example, the message payload includes an encoded value.

```
{
    encoded_temp: "eyAidGVtcGVyYXR1cmUiOiAzMyB9Cg=="
}
```

The decode function in this SQL statement decodes the value in the message payload.

```
SELECT decode(encoded_temp,"base64").temperature AS temp from 'topic/subtopic'
```

Decoding the encoded_temp value results in the following valid JSON document, which allows the SELECT statement to read the temperature value.

```
{ "temperature": 33 }
```

The result of the SELECT statement in this example is shown here.

```
{ "temp": 33 }
```

If the decoded value was not a valid JSON document, the decoded value would be returned as a string.

Decoding protobuf message payload

You can use the decode SQL function to configure a Rule that can decode your protobuf message payload. For more information, see Decoding protobuf message payloads.

> **Important**
>
> If you omit the source-arn or source-account when setting permissions for an AWS IoT principal, any AWS account can invoke your Decode function through other AWS IoT

rules. To secure your function, see Bucket policies in the *Amazon Simple Storage Service User Guide*.

The function signature looks like the following:

```
decode(<ENCODED DATA>, 'proto', '<S3 BUCKET NAME>', '<S3 OBJECT KEY>', '<PROTO NAME>',
 '<MESSAGE TYPE>')
```

`ENCODED DATA`

Specifies the protobuf-encoded data to be decoded. If the entire message sent to the Rule is protobuf-encoded data, you can reference the raw binary incoming payload using `*`. Otherwise, this field must be a base-64 encoded JSON string and a reference to the string can be passed in directly.

1) To decode a raw binary protobuf incoming payload:

```
decode(*, 'proto', ...)
```

2) To decode a protobuf-encoded message represented by a base64-encoded string 'a.b':

```
decode(a.b, 'proto', ...)
```

`proto`

Specifies the data to be decoded in a protobuf message format. If you specify `base64` instead of `proto`, this function will decode base64-encoded strings as JSON.

`S3 BUCKET NAME`

The name of the Amazon S3 bucket where you've uploaded your `FileDescriptorSet` file.

`S3 OBJECT KEY`

The object key that specifies the `FileDescriptorSet` file within the Amazon S3 bucket.

`PROTO NAME`

The name of the `.proto` file (excluding the extension) from which the `FileDescriptorSet` file was generated.

`MESSAGE TYPE`

The name of the protobuf message structure within the `FileDescriptorSet` file, to which the data to be decoded should conform.

An example SQL expression using the decode SQL function can look like the following:

```
SELECT VALUE decode(*, 'proto', 's3-bucket', 'messageformat.desc', 'myproto',
 'messagetype') FROM 'some/topic'
```

- `*`

  Represents a binary incoming payload, which conforms to the protobuf message type called `mymessagetype`.
- `messageformat.desc`

  The `FileDescriptorSet` file stored in an Amazon S3 bucket named `s3-bucket`.
- `myproto`

  The original `.proto` file used to generate the `FileDescriptorSet` file named `myproto.proto`.
- `messagetype`

  The message type called `messagetype` (along with any imported dependencies) as defined in `myproto.proto`.

encode(value, encodingScheme)

Use the `encode` function to encode the payload, which potentially might be non-JSON data, into its string representation based on the encoding scheme. Supported by SQL version 2016-03-23 and later.

value

Any of the valid expressions, as defined in [AWS IoT SQL reference](). You can specify * to encode the entire payload, regardless of whether it's in JSON format. If you supply an expression, the result of the evaluation is converted to a string before it is encoded.

encodingScheme

A literal string representing the encoding scheme you want to use. Currently, only `'base64'` is supported.

endswith(String, String)

Returns a `Boolean` indicating whether the first `String` argument ends with the second `String` argument. If either argument is `Null` or `Undefined`, the result is `Undefined`. Supported by SQL version 2015-10-08 and later.

Example: `endswith("cat","at")` = true.

Argument type 1	Argument type 2	Result
`String`	`String`	True if the first argument ends second argument. Otherwise, f
Other value	Other value	Both arguments are converted using the standard conversion r if the first argument ends in the argument. Otherwise, false. If e argument is `Null` or `Undefine` result is `Undefined` .

exp(Decimal)

Returns e raised to the `Decimal` argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `exp(1)` = e.

Argument type	Result
`Int`	`Decimal` (with double precision), e ^ argument.

Argument type	Result
`Decimal`	`Decimal` (with double precision), e ^ argument.
`String`	`Decimal` (with double precision), e ^ argument. If the `String` cannot be converted to a `Decimal`, the result is `Undefined`.
Other value	`Undefined`.

floor(Decimal)

Rounds the given `Decimal` down to the nearest `Int`. Supported by SQL version 2015-10-08 and later.

Examples:

`floor(1.2)` = 1

`floor(-1.2)` = -2

Argument type	Result
`Int`	`Int`, the argument value.
`Decimal`	`Int`, the `Decimal` value rounded down to the nearest `Int`.
`String`	`Int`. The string is converted to `Decimal` and rounded down to the nearest `Int`. If the string cannot be converted to a `Decimal`, the result is `Undefined`.
Other value	`Undefined`.

get

Extracts a value from a collection-like type (Array, String, Object). No conversion is applied to the first argument. Conversion applies as documented in the table to the second argument. Supported by SQL version 2015-10-08 and later.

Examples:

`get(["a", "b", "c"], 1)` ="b"

`get({"a":"b"}, "a")` = "b"

`get("abc", 0)` = "a"

Argument type 1	Argument type 2	Result
Array	Any Type (converted to `Int`)	The item at the 0-based index o `Array` provided by the second (converted to `Int`). If the conve unsuccessful, the result is Unde If the index is outside the boun `Array` (negative or >= array.ler result is `Undefined` .
String	Any Type (converted to `Int`)	The character at the 0-based in string provided by the second a (converted to `Int`). If the conve unsuccessful, the result is Unde If the index is outside the boun string (negative or >= string.ler result is `Undefined` .
Object	`String` (no conversion is applied)	The value stored in the first arg object corresponding to the str provided as the second argume
Other value	Any value	`Undefined` .

get_dynamodb(tableName, partitionKeyName, partitionKeyValue, sortKeyName, sortKeyValue, roleArn)

Retrieves data from a DynamoDB table. `get_dynamodb()` allows you to query a DynamoDB table while a rule is evaluated. You can filter or augment message payloads using data retrieved from DynamoDB. Supported by SQL version 2016-03-23 and later.

`get_dynamodb()` takes the following parameters:

tableName

The name of the DynamoDB table to query.

partitionKeyName

The name of the partition key. For more information, see DynamoDB Keys.

partitionKeyValue

The value of the partition key used to identify a record. For more information, see DynamoDB Keys.

sortKeyName

(Optional) The name of the sort key. This parameter is required only if the DynamoDB table queried uses a composite key. For more information, see DynamoDB Keys.

sortKeyValue

(Optional) The value of the sort key. This parameter is required only if the DynamoDB table queried uses a composite key. For more information, see DynamoDB Keys.

roleArn

The ARN of an IAM role that grants access to the DynamoDB table. The rules engine assumes this role to access the DynamoDB table on your behalf. Avoid using an overly permissive role. Grant the role only those permissions required by the rule. The following is an example policy that grants access to one DynamoDB table.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": "dynamodb:GetItem",
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/table-name"
        }
    ]
}
```

As an example of how to use `get_dynamodb()`, say you have a DynamoDB table that contains device ID and location information for all of your devices connected to AWS IoT. The following SELECT statement uses the `get_dynamodb()` function to retrieve the location for the specified device ID:

```
SELECT *, get_dynamodb("InServiceDevices", "deviceId", id, "arn:aws:iam::12345678910:role/getdynamo").location AS location FROM 'some/topic'
```

> **Note**
>
> - You can call `get_dynamodb()` a maximum of one time per SQL statement. Calling `get_dynamodb()` multiple times in a single SQL statement causes the rule to terminate without invoking any actions.

get_mqtt_property(name)

References any of the following MQTT5 headers: `contentType`, `payLoadFormatIndicator`, `responseTopic`, and `correlationData`. This function takes any of the following literal strings as an argument: `content_type`, `format_indicator`, `response_topic`, and `correlation_data`. For more information, see the following **Function arguments** table.

contentType

String: A UTF-8 encoded string that describes the content of the publishing message.

payLoadFormatIndicator

String: An Enum string value that indicates whether the payload is formatted as UTF-8. Valid values are `UNSPECIFIED_BYTES` and `UTF8_DATA`.

responseTopic

String: A UTF-8 encoded string that's used as the topic name for a response message. The response topic is used to describe the topic that the receiver should publish to as part of the request-response flow. The topic must not contain wildcard characters.

correlationData

String: The base64-encoded binary data used by the sender of the Request Message to identify which request the Response Message is for when it's received.

The following table shows the acceptable function arguments and their associated return types for the `get_mqtt_property` function:

Function arguments

SQL	Returned data type (if present)	Returned data type (if not present)
`get_mqtt_property("format_indicator")`	String (UNSPECIFIED_BYTES or UTF8_DATA)	String (UNSPECIFIED_BYTES)
`get_mqtt_property("content_type")`	String	Undefined
`get_mqtt_property("response_topic")`	String	Undefined
`get_mqtt_property("correlation_data")`	base64 encoded String	Undefined
`get_mqtt_property("some_invalid_name")`	Undefined	Undefined

The following example Rules SQL references any of the following MQTT5 headers: `contentType`, `payLoadFormatIndicator`, `responseTopic`, and `correlationData`.

```
SELECT *, get_mqtt_property('content_type') as contentType,
```

```
          get_mqtt_property('format_indicator') as payloadFormatIndicator,
          get_mqtt_property('response_topic') as responseTopic,
          get_mqtt_property('correlation_data') as correlationData
FROM 'some/topic'
```

get_or_default(expression, defaultValue)

Returns the default value in the second parameter if specified or else returns undefined, when the expression in the first parameter returns null, undefined, or fails. Supported by SQL version 2016-03-23 and later.

> **Important**
>
> `get_or_default` does not support non-JSON payloads directly as is. If you are using a non-JSON payload, use the `encode` or `decode` functions.

`get_or_default()` takes the following parameters:

expression

Any valid expression containing Data types, Functions, Literals, Variables, Nested object queries, or JSON extensions.

defaultValue

(Optional) Any valid expression containing Data types, Functions, Literals, Variables, Nested object queries, or JSON extensions. This is the value to be returned whenever the first argument returns null, undefined, or fails.

> **Note**
>
> Functions that fetch data from customer owned resources, such as get_secret, get_dynamodb, aws_lambda, get_thing_shadow, decode-protobuf, and machinelearning_predict, are not allowed for the defaultValue parameter.

The following table shows acceptable function arguments for each argument and their associated outputs:

First argument	Second argument	Output
Successful evaluation	Any value or not specified	The first argument value.
Undefined, Null, or failure	Any value including Undefined or Null	The second argument value.
Undefined, Null, or failure	not specified	Undefined

Examples:

Example 1:

The following example provides a defaultValue value if a DynamoDB table or query fails:

```
SELECT
    device_id,
    get_or_default(
        get_dynamodb("DeviceConfig", "deviceId", nonExistentId,
 "arn:aws:iam::123456789012:role/ROLE_NAME"),
        {"mode": "standard", "timeout": 30, "enabled": true }
    ) as config
FROM 'device/telemetry'
```

Example 2:

The following example provides a safe default value "UNKNOWN" if status is undefined:

```
SELECT
  get_or_default( CASE status
    WHEN 'active' THEN 'GOOD'
    WHEN 'inactive' THEN 'BAD'/
    ELSE 'UNKNOWN'
  END, 'UNKNOWN') as status_category
FROM 'topic/subtopic'
```

Example 3:

The following example shows how you can also use get_or_default with a single parameter. This is useful in scenarios where you may not have a clear default value, but you do not want your rule execution to fail.

```
SELECT
  get_dynamodb("DeviceConfig", "deviceId", nonExistentId,
 "arn:aws:iam::123456789012:role/ROLE_NAME") as config
FROM 'device/telemetry'
```

If the DynamoDB lookup fails, the rule execution will fail, and no actions will be executed. If the following SQL is used instead:

```
SELECT
  get_or_default(get_dynamodb("DeviceConfig", "deviceId", nonExistentId,
 "arn:aws:iam::123456789012:role/ROLE_NAME")) as config
FROM 'device/telemetry'
```

The get_or_default statement will evaluate to `Undefined`,so in this example the SELECT statement overall will evaluate to `{}` and any rule actions will be attempted.

Important

We recommend following these best practices to maintain security when using this function:

- Avoid using hardcoded secrets in rule definitions including default values
- Use AWS Secrets Manager for managing sensitive information

get_registry_data(registryAPI, thingName, roleArn)

Retrieves AWS IoT thing registry data in an AWS IoT rule. You can read registry data (such as attributes, thing type, and thing groups a device belongs to) and use this information to filter, enrich, or dynamically route messages. Supported by SQL version 2016-03-23 and later.

`get_registry_data()` takes the following parameters:

registryAPI

The registry API being called. Valid values are `DescribeThing` and `ListThingGroupsForThing`. These values must be constant strings.

thingName

String: The name of the thing whose registry data you want to retrieve.

roleArn

String: A role ARN with `iot:DescribeThing` permission and/or `iot:ListThingGroupsForThing` permission based on the API being called.

The response format of the `get_registry_data` function is the same as the registry API called. For more information, see the DescribeThing and ListThingGroupsForThing APIs.

Example:

You can retrieve thing type information to allow filtering the AWS IoT Core lifecycle event messages for things (with the thing name matching the MQTT client id) where thing type is `testenv`.

```
SELECT *
FROM '$aws/events/lifecycle/+'
WHERE
    get_registry_data("DescribeThing",clientId,[roleArn]).thingTypeName='testenv'
```

Example:

You can retrieve thing attributes for a device with thing name `sensor1` for all messages sent by its gateway device `gateway1`.

```
SELECT *, get_registry_data("DescribeThing","sensor1",
[roleArn]).attributes.temperature_threhold AS device1_tempthreshold
FROM home1/gateway1/sensor1/#
```

Note

- You can call `get_registry_data()` a maximum of one time per SQL statement and substitution templates for actions and error actions.

get_secret(secretId, secretType, key, roleArn)

Retrieves the value of the encrypted `SecretString` or `SecretBinary` field of the current version of a secret in AWS Secrets Manager. For more information about creating and maintaining secrets, see CreateSecret, UpdateSecret, and PutSecretValue.

`get_secret()` takes the following parameters:

secretId

String: The Amazon Resource Name (ARN) or the friendly name of the secret to retrieve.

secretType

String: The secret type. Valid values: `SecretString` **|** `SecretBinary`.

SecretString

- For secrets that you create as JSON objects by using the APIs, the AWS CLI, or the AWS Secrets Manager console:
  - If you specify a value for the `key` parameter, this function returns the value of the specified key.
  - If you don't specify a value for the `key` parameter, this function returns the entire JSON object.
- For secrets that you create as non-JSON objects by using the APIs or the AWS CLI:
  - If you specify a value for the `key` parameter, this function fails with an exception.
  - If you don't specify a value for the `key` parameter, this function returns the contents of the secret.

SecretBinary

- If you specify a value for the `key` parameter, this function fails with an exception.
- If you don't specify a value for the `key` parameter, this function returns the secret value as a base64-encoded UTF-8 string.

key

(Optional) String: The key name inside a JSON object stored in the `SecretString` field of a secret. Use this value when you want to retrieve only the value of a key stored in a secret instead of the entire JSON object.

If you specify a value for this parameter and the secret doesn't contain a JSON object inside its `SecretString` field, this function fails with an exception.

roleArn

String: A role ARN with `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` permissions.

> **Note**
>
> This function always returns the current version of the secret (the version with the `AWSCURRENT` tag). The AWS IoT rules engine caches each secret for up to 15 minutes. As a result, the rules engine can take up to 15 minutes to update a secret. This means that if you retrieve a secret up to 15 minutes after an update with AWS Secrets Manager, this function might return the previous version.
> This function is not metered, but AWS Secrets Manager charges apply. Because of the secret caching mechanism, the rules engine occasionally calls AWS Secrets Manager. Because the rules engine is a fully distributed service, you might see multiple Secrets Manager API calls from the rules engine during the 15-minute caching window.

Examples:

You can use the `get_secret` function in an authentication header in an HTTPS rule action, as in the following API key authentication example.

```
"API_KEY": "${get_secret('API_KEY', 'SecretString', 'API_KEY_VALUE',
 'arn:aws:iam::12345678910:role/getsecret')}"
```

For more information about the HTTPS rule action, see the section called “HTTP”.

get_thing_shadow(thingName, shadowName, roleArn)

Returns the specified shadow of the specified thing. Supported by SQL version 2016-03-23 and later.

thingName

String: The name of the thing whose shadow you want to retrieve.

shadowName

(Optional) String: The name of the shadow. This parameter is required only when referencing named shadows.

roleArn

String: A role ARN with `iot:GetThingShadow` permission.

Examples:

When used with a named shadow, provide the `shadowName` parameter.

```
SELECT * from 'topic/subtopic'
WHERE
    get_thing_shadow("MyThing","MyThingShadow","arn:aws:iam::123456789012:role/
AllowsThingShadowAccess")
    .state.reported.alarm = 'ON'
```

When used with an unnamed shadow, omit the `shadowName` parameter.

```
SELECT * from 'topic/subtopic'
WHERE
    get_thing_shadow("MyThing","arn:aws:iam::123456789012:role/
AllowsThingShadowAccess")
    .state.reported.alarm = 'ON'
```

get_user_properties(userPropertyKey)

References User Properties, which is one type of property headers supported in MQTT5.

userProperty

String: A user property is a key-value pair. This function takes the key as an argument and returns an array of all values that match the associated key.

Function arguments

For the following User Properties in the message headers:

Key	Value
some key	some value
a different key	a different value
some key	value with duplicate key

The following table shows the expected SQL behavior:

SQL	Returned data type	Returned data value
get_user_properties('some key')	Array of String	`['some value', 'value with duplicate key']`
get_user_properties('other key')	Array of String	`['a different value']`
get_user_properties( )	Array of key-value pair Objects	`[{'"some key": "some value"'}, {"other key": "a different value"}, {"some key": "value with duplicate key"}]`
get_user_properties('non-existent key')	Undefined	

The following example Rules SQL references User Properties (a type of MQTT5 property header) into the payload:

```
SELECT *, get_user_properties('user defined property key') as userProperty
FROM 'some/topic'
```

Hashing functions

AWS IoT provides the following hashing functions:

- md2
- md5
- sha1
- sha224
- sha256
- sha384
- sha512

All hash functions expect one string argument. The result is the hashed value of that string. Standard string conversions apply to non-string arguments. All hash functions are supported by SQL version 2015-10-08 and later.

Examples:

`md2("hello")` = "a9046c73e00331af68917d3804f70655"

`md5("hello")` = "5d41402abc4b2a76b9719d911017c592"

indexof(String, String)

Returns the first index (0-based) of the second argument as a substring in the first argument. Both arguments are expected as strings. Arguments that are not strings are subjected to standard string conversion rules. This function does not apply to arrays, only to strings. Supported by SQL version 2016-03-23 and later.

Examples:

`indexof("abcd", "bc")` = 1

isNull()

Returns true if the argument is the `Null` value. Supported by SQL version 2016-03-23 and later.

Examples:

`isNull(5)` = false.

`isNull(Null)` = true.

Argument type	Result
`Int`	false
`Decimal`	false
`Boolean`	false
`String`	false
`Array`	false
`Object`	false
`Null`	true
`Undefined`	false

isUndefined()

Returns true if the argument is `Undefined`. Supported by SQL version 2016-03-23 and later.

Examples:

`isUndefined(5)` = false.

`isUndefined(floor([1,2,3]))` = true.

Argument type	Result
`Int`	false
`Decimal`	false
`Boolean`	false
`String`	false
`Array`	false

Argument type	Result
`Object`	false
`Null`	false
`Undefined`	true

length(String)

Returns the number of characters in the provided string. Standard conversion rules apply to non-`String` arguments. Supported by SQL version 2016-03-23 and later.

Examples:

`length("hi")` = 2

`length(false)` = 5

ln(Decimal)

Returns the natural logarithm of the argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `ln(e)` = 1.

Argument type	Result
`Int`	`Decimal` (with double precision), the natural log of the argument.
`Decimal`	`Decimal` (with double precision), the natural log of the argument.
`Boolean`	`Undefined` .
`String`	`Decimal` (with double precision), the natural log of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined` .

Argument type	Result
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

log(Decimal)

Returns the base 10 logarithm of the argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `log(100)` = 2.0.

Argument type	Result
`Int`	`Decimal` (with double precision), the base 10 log of the argument.
`Decimal`	`Decimal` (with double precision), the base 10 log of the argument.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the base 10 log of the argument. If the `String` cannot be converted to a `Decimal`, the result is `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

lower(String)

Returns the lowercase version of the given `String`. Non-string arguments are converted to strings using the standard conversion rules. Supported by SQL version 2015-10-08 and later.

Examples:

`lower("HELLO")` = "hello".

`lower(["HELLO"])` = "[\"hello\"]".

lpad(String, Int)

Returns the `String` argument, padded on the left side with the number of spaces specified by the second argument. The `Int` argument must be between 0 and 1000. If the provided value is outside of this valid range, the argument is set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-08 and later.

Examples:

`lpad("hello", 2)` = " hello".

`lpad(1, 3)` = "   1"

Argument type 1	Argument type 2	Result
`String`	`Int`	`String`, the provided `String` the left side with a number of s to the provided `Int`.
`String`	`Decimal`	The `Decimal` argument is roun to the nearest `Int` and the `Str` padded on the left with the spe number of spaces.
`String`	`String`	The second argument is conver `Decimal`, which is rounded dov nearest `Int`, and the `String` is with the specified number spac

Argument type 1	Argument type 2	Result
		the left. If the second argumen be converted to an Int, the res Undefined .
Other value	Int/Decimal/String	The first value is converted to a using the standard conversions the LPAD function is applied or String. If it cannot be convert result is Undefined .
Any value	Other value	Undefined .

ltrim(String)

Removes all leading white space (tabs and spaces) from the provided String. Supported by SQL version 2015-10-08 and later.

Example:

`Ltrim(" h i ")` = "hi ".

Argument type	Result
Int	The String representation of the Int with all leading white space removed.
Decimal	The String representation of the Decimal with all leading white space removed.
Boolean	The String representation of the Boolean ("true" or "false") with all leading white space removed.
String	The argument with all leading white space removed.

Argument type	Result
Array	The `String` representation of the `Array` (using standard conversion rules) with all leading white space removed.
Object	The `String` representation of the Object (using standard conversion rules) with all leading white space removed.
Null	`Undefined` .
Undefined	`Undefined` .

machinelearning_predict(modelId, roleArn, record)

Use the `machinelearning_predict` function to make predictions using the data from an MQTT message based on an Amazon SageMaker AI model. Supported by SQL version 2015-10-08 and later. The arguments for the `machinelearning_predict` function are:

modelId

The ID of the model against which to run the prediction. The real-time endpoint of the model must be enabled.

roleArn

The IAM role that has a policy with `machinelearning:Predict` and `machinelearning:GetMLModel` permissions and allows access to the model against which the prediction is run.

record

The data to be passed into the SageMaker AI Predict API. This should be represented as a single layer JSON object. If the record is a multi-level JSON object, the record is flattened by serializing its values. For example, the following JSON:

```
{ "key1": {"innerKey1": "value1"}, "key2": 0}
```

would become:

```
{ "key1": "{\"innerKey1\": \"value1\"}", "key2": 0}
```

The function returns a JSON object with the following fields:

predictedLabel

The classification of the input based on the model.

details

Contains the following attributes:

PredictiveModelType

The model type. Valid values are REGRESSION, BINARY, MULTICLASS.

Algorithm

The algorithm used by SageMaker AI to make predictions. The value must be SGD.

predictedScores

Contains the raw classification score corresponding to each label.

predictedValue

The value predicted by SageMaker AI.

mod(Decimal, Decimal)

Returns the remainder of the division of the first argument by the second argument. Equivalent to remainder(Decimal, Decimal). You can also use "%" as an infix operator for the same modulo functionality. Supported by SQL version 2015-10-08 and later.

Example: `mod(8, 3)` = 2.

Left operand	Right operand	Output
`Int`	`Int`	`Int`, the first argument modulo argument.
`Int/Decimal`	`Int/Decimal`	`Decimal`, the first argument m second operand.

Left operand	Right operand	Output
`String/Int/Decimal`	`String/Int/Decimal`	If all strings convert to decimal is the first argument modulo th argument. Otherwise, `Undefin`
Other value	Other value	`Undefined` .

nanvl(AnyValue, AnyValue)

Returns the first argument if it is a valid `Decimal`. Otherwise, the second argument is returned. Supported by SQL version 2015-10-08 and later.

Example: `Nanvl(8, 3)` = 8.

Argument type 1	Argument type 2	Output
Undefined	Any value	The second argument.
Null	Any value	The second argument.
`Decimal` (NaN)	Any value	The second argument.
`Decimal` (not NaN)	Any value	The first argument.
Other value	Any value	The first argument.

newuuid()

Returns a random 16-byte UUID. Supported by SQL version 2015-10-08 and later.

Example: `newuuid()` = `123a4567-b89c-12d3-e456-789012345000`

numbytes(String)

Returns the number of bytes in the UTF-8 encoding of the provided string. Standard conversion rules apply to non-`String` arguments. Supported by SQL version 2016-03-23 and later.

Examples:

`numbytes("hi")` = 2

`numbytes("€")` = 3

parse_time(String, Long[, String])

Use the `parse_time` function to format a timestamp into a human-readable date/time format. Supported by SQL version 2016-03-23 and later. To convert a timestamp string into milliseconds, see time_to_epoch(String, String).

The `parse_time` function expects the following arguments:

pattern

(String) A date/time pattern that follows Joda-Time formats.

timestamp

(Long) The time to be formatted in milliseconds since Unix epoch. See function timestamp().

timezone

(String) The time zone of the formatted date/time. The default is "UTC". The function supports Joda-Time time zones. This argument is optional.

Examples:

When this message is published to the topic 'A/B', the payload `{"ts": "1970.01.01 AD at 21:46:40 CST"}` is sent to the S3 bucket:

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT parse_time(\"yyyy.MM.dd G 'at' HH:mm:ss z\", 100000000,
 'America/Belize' ) as ts FROM 'A/B'",

        "ruleDisabled": false,
        "awsIotSqlVersion": "2016-03-23",
        "actions": [
            {
                "s3": {
                    "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
                    "bucketName": "BUCKET_NAME",
                    "key": "KEY_NAME"
```

```
                }
            }
        ],
        "ruleName": "RULE_NAME"
    }
}
```

When this message is published to the topic 'A/B', a payload similar to `{"ts": "2017.06.09 AD at 17:19:46 UTC"}` (but with the current date/time) is sent to the S3 bucket:

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT parse_time(\"yyyy.MM.dd G 'at' HH:mm:ss z\", timestamp() ) as ts
 FROM 'A/B'",
        "awsIotSqlVersion": "2016-03-23",
        "ruleDisabled": false,
        "actions": [
            {
                "s3": {
                    "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
                    "bucketName": "BUCKET_NAME",
                    "key": "KEY_NAME"
                }
            }
        ],
        "ruleName": "RULE_NAME"
    }
}
```

`parse_time()` can also be used as a substitution template. For example, when this message is published to the topic 'A/B', the payload is sent to the S3 bucket with key = "2017":

```
{
    "ruleArn": "arn:aws:iot:us-east-2:ACCOUNT_ID:rule/RULE_NAME",
    "topicRulePayload": {
        "sql": "SELECT * FROM 'A/B'",
        "awsIotSqlVersion": "2016-03-23",
        "ruleDisabled": false,
        "actions": [{
            "s3": {
                "roleArn": "arn:aws:iam::ACCOUNT_ID:rule:role/ROLE_NAME",
```

```
                "bucketName": "BUCKET_NAME",
                "key": "${parse_time('yyyy', timestamp(), 'UTC')}"
            }
        }],
        "ruleName": "RULE_NAME"
    }
}
```

power(Decimal, Decimal)

Returns the first argument raised to the second argument. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later. Supported by SQL version 2015-10-08 and later.

Example: `power(2, 5)` = 32.0.

Argument type 1	Argument type 2	Output
`Int/Decimal`	`Int/Decimal`	A `Decimal` (with double precis the first argument raised to the argument's power.
`Int/Decimal/String`	`Int/Decimal/String`	A `Decimal` (with double precis the first argument raised to the argument's power. Any strings a converted to decimals. If any St to be converted to `Decimal`, th `Undefined` .
Other value	Other value	`Undefined` .

principal()

Returns the principal that the device uses for authentication, based on how the triggering message was published. The following table describes the principal returned for each publishing method and protocol.

How the message is published	Protocol	Credential type
MQTT client	MQTT	X.509 device certificate
AWS IoT console MQTT client	MQTT	IAM user or role
AWS CLI	HTTP	IAM user or role
AWS IoT Device SDK	MQTT	X.509 device certificate
AWS IoT Device SDK	MQTT over WebSocket	IAM user or role

The following examples show the different types of values that `principal()` can return:

- X.509 certificate thumbprint: `ba67293af50bf2506f5f93469686da660c7c844e7b3950bfb16813e0d31e9373`
- IAM role ID and session name: `ABCD1EFG3HIJK2LMNOP5:my-session-name`
- Returns a user ID: `ABCD1EFG3HIJK2LMNOP5`

rand()

Returns a pseudorandom, uniformly distributed double between 0.0 and 1.0. Supported by SQL version 2015-10-08 and later.

Example:

`rand()` = 0.8231909191640703

regexp_matches(String, String)

Returns true if the string (first argument) contains a match for the regular expression (second argument). If you use `|` in the regular expression, use it with `()`.

Examples:

`regexp_matches("aaaa", "a{2,}")` = true.

`regexp_matches("aaaa", "b")` = false.

`regexp_matches("aaa", "(aaa|bbb)")` = true.

`regexp_matches("bbb", "(aaa|bbb)")` = true.

`regexp_matches("ccc", "(aaa|bbb)")` = false.

First argument:

Argument type	Result
`Int`	The `String` representation of the `Int`.
`Decimal`	The `String` representation of the `Decimal`.
`Boolean`	The `String` representation of the Boolean ("true" or "false").
`String`	The `String`.
Array	The `String` representation of the `Array` (using standard conversion rules).
Object	The `String` representation of the Object (using standard conversion rules).
Null	`Undefined` .
Undefined	`Undefined` .

Second argument:

Must be a valid regex expression. Non-string types are converted to `String` using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not valid regex, the result is `Undefined`.

regexp_replace(String, String, String)

Replaces all occurrences of the second argument (regular expression) in the first argument with the third argument. Reference capture groups with "$". Supported by SQL version 2015-10-08 and later.

Example:

`regexp_replace("abcd", "bc", "x")` = "axd".

`regexp_replace("abcd", "b(.*)d", "$1")` = "ac".

First argument:

Argument type	Result
`Int`	The `String` representation of the `Int`.
`Decimal`	The `String` representation of the `Decimal`.
`Boolean`	The `String` representation of the Boolean ("true" or "false").
`String`	The source value.
Array	The `String` representation of the `Array` (using standard conversion rules).
Object	The `String` representation of the Object (using standard conversion rules).
Null	`Undefined` .
Undefined	`Undefined` .

Second argument:

Must be a valid regex expression. Non-string types are converted to `String` using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not a valid regex expression, the result is `Undefined`.

Third argument:

Must be a valid regex replacement string. (Can reference capture groups.) Non-string types are converted to `String` using the standard conversion rules. If the (converted) argument is not a valid regex replacement string, the result is `Undefined`.

regexp_substr(String, String)

Finds the first match of the second parameter (regex) in the first parameter. Reference capture groups with "$". Supported by SQL version 2015-10-08 and later.

Example:

`regexp_substr("hihihello", "hi")` = "hi"

`regexp_substr("hihihello", "(hi)*")` = "hihi"

First argument:

Argument type	Result
`Int`	The `String` representation of the `Int`.
`Decimal`	The `String` representation of the `Decimal`.
`Boolean`	The `String` representation of the Boolean ("true" or "false").
`String`	The `String` argument.
Array	The `String` representation of the `Array` (using standard conversion rules).
Object	The `String` representation of the Object (using standard conversion rules).
Null	`Undefined` .
Undefined	`Undefined` .

Second argument:

Must be a valid regex expression. Non-string types are converted to `String` using the standard conversion rules. Depending on the type, the resultant string might not be a valid regular expression. If the (converted) argument is not a valid regex expression, the result is `Undefined`.

remainder(Decimal, Decimal)

Returns the remainder of the division of the first argument by the second argument. Equivalent to mod(Decimal, Decimal). You can also use "%" as an infix operator for the same modulo functionality. Supported by SQL version 2015-10-08 and later.

Example: `remainder(8, 3)` = 2.

Left operand	Right operand	Output
`Int`	`Int`	`Int`, the first argument modulo argument.
`Int/Decimal`	`Int/Decimal`	`Decimal`, the first argument m second operand.
`String/Int/Decimal`	`String/Int/Decimal`	If all strings convert to decimal is the first argument modulo th argument. Otherwise, `Undefir`
Other value	Other value	`Undefined` .

replace(String, String, String)

Replaces all occurrences of the second argument in the first argument with the third argument. Supported by SQL version 2015-10-08 and later.

Example:

`replace("abcd", "bc", "x")` = `"axd"`.

`replace("abcdabcd", "b", "x")` = `"axcdaxcd"`.

All arguments

Argument type	Result
`Int`	The `String` representation of the `Int`.

Argument type	Result
`Decimal`	The `String` representation of the `Decimal`.
`Boolean`	The `String` representation of the Boolean ("true" or "false").
`String`	The source value.
Array	The `String` representation of the `Array` (using standard conversion rules).
Object	The `String` representation of the Object (using standard conversion rules).
Null	`Undefined` .
Undefined	`Undefined` .

rpad(String, Int)

Returns the string argument, padded on the right side with the number of spaces specified in the second argument. The `Int` argument must be between 0 and 1000. If the provided value is outside of this valid range, the argument is set to the nearest valid value (0 or 1000). Supported by SQL version 2015-10-08 and later.

Examples:

`rpad("hello", 2)` = `"hello  "`.

`rpad(1, 3)` = `"1   "`.

Argument type 1	Argument type 2	Result
`String`	`Int`	The `String` is padded on the right side with a number of spaces

Argument type 1	Argument type 2	Result
		equal to the provided `Int`.
`String`	`Decimal`	The `Decimal` argument is rounded down to the nearest `Int` and the string is padded on the right side with a number of spaces equal to the provided `Int`.
`String`	`String`	The second argument is converted to a `Decimal`, which is rounded down to the nearest `Int`. The `String` is padded on the right side with a number of spaces equal to the `Int` value.
Other value	`Int/Decimal/String`	The first value is converted to a `String` using the standard conversions, and the rpad function is applied on that `String`. If it cannot be converted, the result is `Undefined` .
Any value	Other value	`Undefined` .

round(Decimal)

Rounds the given `Decimal` to the nearest `Int`. If the `Decimal` is equidistant from two `Int` values (for example, 0.5), the `Decimal` is rounded up. Supported by SQL version 2015-10-08 and later.

Example: `Round(1.2)` = 1.

`Round(1.5)` = 2.

`Round(1.7)` = 2.

`Round(-1.1)` = -1.

`Round(-1.5)` = -2.

Argument type	Result
`Int`	The argument.
`Decimal`	`Decimal` is rounded down to the nearest `Int`.
`String`	`Decimal` is rounded down to the nearest `Int`. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Other value	`Undefined` .

rtrim(String)

Removes all trailing white space (tabs and spaces) from the provided `String`. Supported by SQL version 2015-10-08 and later.

Examples:

`rtrim(" h i ")` = " h i"

Argument type	Result
`Int`	The `String` representation of the `Int`.

Argument type	Result
`Decimal`	The `String` representation of the `Decimal`.
`Boolean`	The `String` representation of the Boolean ("true" or "false").
Array	The `String` representation of the `Array` (using standard conversion rules).
Object	The `String` representation of the Object (using standard conversion rules).
Null	`Undefined` .
Undefined	`Undefined`

sign(Decimal)

Returns the sign of the given number. When the sign of the argument is positive, 1 is returned. When the sign of the argument is negative, -1 is returned. If the argument is 0, 0 is returned. Supported by SQL version 2015-10-08 and later.

Examples:

`sign(-7)` = -1.

`sign(0)` = 0.

`sign(13)` = 1.

Argument type	Result
`Int`	`Int`, the sign of the `Int` value.
`Decimal`	`Int`, the sign of the `Decimal` value.
`String`	`Int`, the sign of the `Decimal` value. The string is converted to a `Decimal`

Argument type	Result
	value, and the sign of the `Decimal` value is returned. If the `String` cannot be converted to a `Decimal`, the result is `Undefined` . Supported by SQL version 2015-10-08 and later.
Other value	`Undefined` .

sin(Decimal)

Returns the sine of a number in radians. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `sin(0)` = 0.0

Argument type	Result
`Int`	`Decimal` (with double precision), the sine of the argument.
`Decimal`	`Decimal` (with double precision), the sine of the argument.
`Boolean`	`Undefined` .
`String`	`Decimal` (with double precision), the sine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
`Undefined`	`Undefined` .

sinh(Decimal)

Returns the hyperbolic sine of a number. `Decimal` values are rounded to double precision before function application. The result is a `Decimal` value of double precision. Supported by SQL version 2015-10-08 and later.

Example: `sinh(2.3)` = 4.936961805545957

Argument type	Result
`Int`	`Decimal` (with double precision), the hyperbolic sine of the argument.
`Decimal`	`Decimal` (with double precision), the hyperbolic sine of the argument.
`Boolean`	`Undefined`.
`String`	`Decimal` (with double precision), the hyperbolic sine of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined`.
Array	`Undefined`.
Object	`Undefined`.
Null	`Undefined`.
Undefined	`Undefined`.

sourceip()

Retrieves the IP address of a device or the router that connects to it. If your device is connected to the internet directly, the function will return the source IP address of the device. If your device is connected to a router that connects to the internet, the function will return the source IP address of the router. Supported by SQL version 2016-03-23. `sourceip()` doesn't take any parameters.

> **Important**
>
> A device's public source IP address is often the IP address of the last Network Address Translation (NAT) Gateway such as your internet service provider's router or cable modem.

Examples:

`sourceip()="192.158.1.38"`

`sourceip()="1.102.103.104"`

`sourceip()="2001:db8:ff00::12ab:34cd"`

SQL example:

`SELECT *, sourceip() as deviceIp FROM 'some/topic'`

Examples of how to use the sourceip() function in AWS IoT Core rule actions:

Example 1

The following example shows how to call the () function as a substitution template in a DynamoDB action.

```
{
 "topicRulePayload": {
  "sql": "SELECT * AS message FROM 'some/topic'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
   {
    "dynamoDB": {
     "tableName": "my_ddb_table",
     "hashKeyField": "key",
     "hashKeyValue": "${sourceip()}",
     "rangeKeyField": "timestamp",
     "rangeKeyValue": "${timestamp()}",
     "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB"
    }
   }
  ]
```

```
 }
}
```

Example 2

The following example shows how to add the sourceip() function as an MQTT user property using substitution templates.

```
{
 "topicRulePayload": {
  "sql": "SELECT * FROM 'some/topic'",
  "ruleDisabled": false,
  "awsIotSqlVersion": "2016-03-23",
  "actions": [
   {
    "republish": {
     "topic": "${topic()}/republish",
     "roleArn": "arn:aws:iam::123456789012:role/aws_iot_republish",
     "headers": {
      "payloadFormatIndicator": "UTF8_DATA",
      "contentType": "rule/contentType",
      "correlationData": "cnVsZSBjb3JyZWxhdGlvbiBkYXRh",
      "userProperties": [
       {
        "key": "ruleKey1",
        "value": "ruleValue1"
       },
       {
        "key": "sourceip",
        "value": "${sourceip()}"
       }
      ]
     }
    }
   }
  ]
 }
}
```

You can retrieve the source IP address from messages passing to AWS IoT Core rules from both Message Broker and Basic Ingest pathways. You can also retrieve the source IP for both IPv4 and IPv6 messages. The source IP will be displayed like the following:

IPv6: `yyyy:yyyy:yyyy::yyyy:yyyy`

IPv4: `xxx.xxx.xxx.xxx`

> **Note**
>
> The original source IP won't be passed though Republish action.

substring(String, Int[, Int])

Expects a `String` followed by one or two `Int` values. For a `String` and a single `Int` argument, this function returns the substring of the provided `String` from the provided `Int` index (0-based, inclusive) to the end of the `String`. For a `String` and two `Int` arguments, this function returns the substring of the provided `String` from the first `Int` index argument (0-based, inclusive) to the second `Int` index argument (0-based, exclusive). Indices that are less than zero are set to zero. Indices that are greater than the `String` length are set to the `String` length. For the three argument version, if the first index is greater than (or equal to) the second index, the result is the empty `String`.

If the arguments provided are not (*`String`*, *`Int`*), or (*`String`*, *`Int`*, *`Int`*), the standard conversions are applied to the arguments to attempt to convert them into the correct types. If the types cannot be converted, the result of the function is `Undefined`. Supported by SQL version 2015-10-08 and later.

Examples:

`substring("012345", 0)` = "012345".

`substring("012345", 2)` = "2345".

`substring("012345", 2.745)` = "2345".

`substring(123, 2)` = "3".

`substring("012345", -1)` = "012345".

`substring(true, 1.2)` = "rue".

`substring(false, -2.411E247)` = "false".

`substring("012345", 1, 3)` = "12".

`substring("012345", -50, 50)` = "012345".

`substring("012345", 3, 1)` = "".

sql_version()

Returns the SQL version specified in this rule. Supported by SQL version 2015-10-08 and later.

Example:

`sql_version()` = "2016-03-23"

sqrt(Decimal)

Returns the square root of a number. `Decimal` arguments are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `sqrt(9)` = 3.0.

Argument type	Result
`Int`	The square root of the argument.
`Decimal`	The square root of the argument.
`Boolean`	`Undefined` .
`String`	The square root of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

startswith(String, String)

Returns Boolean, whether the first string argument starts with the second string argument. If either argument is Null or Undefined, the result is Undefined. Supported by SQL version 2015-10-08 and later.

Example:

startswith("ranger","ran") = true

Argument type 1	Argument type 2	Result
String	String	Whether the first string starts w second string.
Other value	Other value	Both arguments are converted using the standard conversion r Returns true if the first string st the second string. If either argu is Null or Undefined , the re Undefined .

tan(Decimal)

Returns the tangent of a number in radians. Decimal values are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: tan(3) = -0.1425465430742778

Argument type	Result
Int	Decimal (with double precision), the tangent of the argument.
Decimal	Decimal (with double precision), the tangent of the argument.
Boolean	Undefined .

Argument type	Result
`String`	`Decimal` (with double precision), the tangent of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Array	`Undefined` .
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

tanh(Decimal)

Returns the hyperbolic tangent of a number in radians. `Decimal` values are rounded to double precision before function application. Supported by SQL version 2015-10-08 and later.

Example: `tanh(2.3)` = 0.9800963962661914

Argument type	Result
`Int`	`Decimal` (with double precision), the hyperbolic tangent of the argument.
`Decimal`	`Decimal` (with double precision), the hyperbolic tangent of the argument.
`Boolean`	`Undefined` .
`String`	`Decimal` (with double precision), the hyperbolic tangent of the argument. If the string cannot be converted to a `Decimal`, the result is `Undefined` .
Array	`Undefined` .

Argument type	Result
Object	`Undefined` .
Null	`Undefined` .
Undefined	`Undefined` .

time_to_epoch(String, String)

Use the `time_to_epoch` function to convert a timestamp string into a number of milliseconds in Unix epoch time. Supported by SQL version 2016-03-23 and later. To convert milliseconds to a formatted timestamp string, see parse_time(String, Long[, String]).

The `time_to_epoch` function expects the following arguments:

timestamp

(String) The timestamp string to be converted to milliseconds since Unix epoch. If the timestamp string doesn't specify a timezone, the function uses the UTC timezone.

pattern

(String) A date/time pattern that follows JDK11 Time Formats.

Examples:

`time_to_epoch("2020-04-03 09:45:18 UTC+01:00", "yyyy-MM-dd HH:mm:ss VV")` = 1585903518000

`time_to_epoch("18 December 2015", "dd MMMM yyyy")` = 1450396800000

`time_to_epoch("2007-12-03 10:15:30.592 America/Los_Angeles", "yyyy-MM-dd HH:mm:ss.SSS z")` = 1196705730592

timestamp()

Returns the current timestamp in milliseconds from 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, as observed by the AWS IoT rules engine. Supported by SQL version 2015-10-08 and later.

Example: `timestamp()` = `1481825251155`

topic(Decimal)

Returns the topic to which the message that triggered the rule was sent. If no parameter is specified, the entire topic is returned. The `Decimal` parameter is used to specify a specific topic segment, with 1 designating the first segment. For the topic `foo/bar/baz`, topic(1) returns `foo`, topic(2) returns `bar`, and so on. Supported by SQL version 2015-10-08 and later.

Examples:

`topic()` = "things/myThings/thingOne"

`topic(1)` = "things"

When Basic Ingest is used, the initial prefix of the topic (`$aws/rules/`*`rule-name`*) is not available to the topic() function. For example, given the topic:

`$aws/rules/BuildingManager/Buildings/Building5/Floor2/Room201/Lights`

`topic()` = "Buildings/Building5/Floor2/Room201/Lights"

`topic(3)` = "Floor2"

traceid()

Returns the trace ID (UUID) of the MQTT message, or `Undefined` if the message wasn't sent over MQTT. Supported by SQL version 2015-10-08 and later.

Example:

`traceid()` = "12345678-1234-1234-1234-123456789012"

transform(String, Object, Array)

Returns an array of objects that contains the result of the specified transformation of the `Object` parameter on the `Array` parameter.

Supported by SQL version 2016-03-23 and later.

String

The transformation mode to use. Refer to the following table for the supported transformation modes and how they create the `Result` from the `Object` and `Array` parameters.

Object

An object that contains the attributes to apply to each element of the `Array`.

Array

An array of objects into which the attributes of `Object` are applied.

Each object in this Array corresponds to an object in the function's response. Each object in the function's response contains the attributes present in the original object and the attributes provided by `Object` as determined by the transformation mode specified in `String`.

`String` parameter	`Object` parameter	`Array` parameter	Result
`enrichArray`	Object	Array of objects	An Array of objects in which each object contains the attributes of an element from the `Array` parameter and the attributes of the `Object` parameter.
Any other value	Any value	Any value	Undefined

> **Note**
> The array returned by this function is limited to 128 KiB.

Transform function example 1

This example shows how the **transform()** function produces a single array of objects from a data object and an array.

In this example, the following message is published to the MQTT topic A/B.

```
{
```

```
    "attributes": {
        "data1": 1,
        "data2": 2
    },
    "values": [
        {
            "a": 3
        },
        {
            "b": 4
        },
        {
            "c": 5
        }
    ]
}
```

This SQL statement for a topic rule action uses the **transform()** function with a `String` value of `enrichArray`. In this example, `Object` is the `attributes` property from the message payload and `Array` is the `values` array, which contains three objects.

```
select value transform("enrichArray", attributes, values) from 'A/B'
```

Upon receiving the message payload, the SQL statement evaluates to the following response.

```
[
  {
    "a": 3,
    "data1": 1,
    "data2": 2
  },
  {
    "b": 4,
    "data1": 1,
    "data2": 2
  },
  {
    "c": 5,
    "data1": 1,
    "data2": 2
  }
]
```

Transform function example 2

This example shows how the **transform()** function can use literal values to include and rename individual attributes from the message payload.

In this example, the following message is published to the MQTT topic A/B. This is the same message that was used in the section called "Transform function example 1".

```
{
    "attributes": {
        "data1": 1,
        "data2": 2
    },
    "values": [
        {
            "a": 3
        },
        {
            "b": 4
        },
        {
            "c": 5
        }
    ]
}
```

This SQL statement for a topic rule action uses the **transform()** function with a `String` value of `enrichArray`. The `Object` in the **transform()** function has a single attribute named `key` with the value of `attributes.data1` in the message payload and `Array` is the `values` array, which contains the same three objects used in the previous example.

```
select value transform("enrichArray", {"key": attributes.data1}, values) from 'A/B'
```

Upon receiving the message payload, this SQL statement evaluates to the following response. Notice how the `data1` property is named `key` in the response.

```
[
  {
    "a": 3,
    "key": 1
  },
  {
```

```
    "b": 4,
    "key": 1
  },
  {
    "c": 5,
    "key": 1
  }
]
```

Transform function example 3

This example shows how the **transform()** function can be used in nested SELECT clauses to select multiple attributes and create new objects for subsequent processing.

In this example, the following message is published to the MQTT topic A/B.

```
{
  "data1": "example",
  "data2": {
    "a": "first attribute",
    "b": "second attribute",
    "c": [
      {
        "x": {
          "someInt": 5,
          "someString": "hello"
        },
        "y": true
      },
      {
        "x": {
          "someInt": 10,
          "someString": "world"
        },
        "y": false
      }
    ]
  }
}
```

The `Object` for this transform function is the object returned by the SELECT statement, which contains the a and b elements of the message's `data2` object. The `Array` parameter consists of the two objects from the `data2.c` array in the original message.

```
select value transform('enrichArray', (select a, b from data2), (select value c from
 data2)) from 'A/B'
```

With the preceding message, the SQL statement evaluates to the following response.

```
[
  {
    "x": {
      "someInt": 5,
      "someString": "hello"
    },
    "y": true,
    "a": "first attribute",
    "b": "second attribute"
  },
  {
    "x": {
      "someInt": 10,
      "someString": "world"
    },
    "y": false,
    "a": "first attribute",
    "b": "second attribute"
  }
]
```

The array returned in this response could be used with topic rule actions that support `batchMode`.

trim(String)

Removes all leading and trailing white space from the provided `String`. Supported by SQL version 2015-10-08 and later.

Example:

`Trim(" hi ")` = "hi"

Argument type	Result
`Int`	The `String` representation of the `Int` with all leading and trailing white space removed.
`Decimal`	The `String` representation of the `Decimal` with all leading and trailing white space removed.
`Boolean`	The `String` representation of the `Boolean` ("true" or "false") with all leading and trailing white space removed.
`String`	The `String` with all leading and trailing white space removed.
Array	The `String` representation of the `Array` using standard conversion rules.
Object	The `String` representation of the Object using standard conversion rules.
Null	`Undefined .`
Undefined	`Undefined .`

trunc(Decimal, Int)

Truncates the first argument to the number of `Decimal` places specified by the second argument. If the second argument is less than zero, it is set to zero. If the second argument is greater than 34, it is set to 34. Trailing zeroes are stripped from the result. Supported by SQL version 2015-10-08 and later.

Examples:

`trunc(2.3, 0)` = 2.

`trunc(2.3123, 2)` = 2.31.

`trunc(2.888, 2)` = 2.88.

`trunc(2.00, 5)` = 2.

Argument type 1	Argument type 2	Result
`Int`	`Int`	The source value.
`Int/Decimal`	`Int/Decimal`	The first argument is truncated length described by the second The second argument, if not an rounded down to the nearest I
`Int/Decimal/String`	`Int/Decimal`	The first argument is truncated length described by the second The second argument, if not an is rounded down to the nearest `String` is converted to a `Deci` If the string conversion fails, th `Undefined` .
Other value		`Undefined` .

upper(String)

Returns the uppercase version of the given `String`. Non-`String` arguments are converted to `String` using the standard conversion rules. Supported by SQL version 2015-10-08 and later.

Examples:

`upper("hello")` = "HELLO"

`upper(["hello"])` = "[\"HELLO\"]"

Literals

You can directly specify literal objects in the SELECT and WHERE clauses of your rule SQL, which can be useful for passing information.

> **Note**
>
> Literals are available only when using SQL version 2016-03-23 or later.

JSON object syntax is used (key-value pairs, comma-separated, where keys are strings and values are JSON values, wrapped in curly brackets {}). For example:

Incoming payload published on topic `topic/subtopic`: `{"lat_long": [47.606,-122.332]}`

SQL statement: `SELECT {'latitude': get(lat_long, 0),'longitude':get(lat_long, 1)} as lat_long FROM 'topic/subtopic'`

The resulting outgoing payload would be: `{"lat_long": {"latitude":47.606,"longitude":-122.332}}`.

You can also directly specify arrays in the SELECT and WHERE clauses of your rule SQL, which allows you to group information. JSON syntax is used (wrap comma-separated items in square brackets [] to create an array literal). For example:

Incoming payload published on topic `topic/subtopic`: `{"lat": 47.696, "long": -122.332}`

SQL statement: `SELECT [lat,long] as lat_long FROM 'topic/subtopic'`

The resulting output payload would be: `{"lat_long": [47.606,-122.332]}`.

Case statements

Case statements can be used for branching execution, like a switch statement.

Syntax:

```
CASE v WHEN t[1] THEN r[1]
  WHEN t[2] THEN r[2] ...
  WHEN t[n] THEN r[n]
  ELSE r[e] END
```

The expression *`v`* is evaluated and matched for equality against the *`t[i]`* value of each `WHEN` clause. If a match is found, the corresponding *`r[i]`* expression becomes the result of the `CASE` statement. The `WHEN` clauses are evaluated in order so that if there's more than one matching clause, the result of the first matching clause becomes the result of the `CASE` statement. If there are no matches, *r[e]* of the `ELSE` clause is the result. If there's no match and no `ELSE` clause, the result is `Undefined`.

`CASE` statements require at least one `WHEN` clause. An `ELSE` clause is optional.

For example:

Incoming payload published on topic `topic/subtopic`:

```
{
    "color":"yellow"
}
```

SQL statement:

```
SELECT CASE color
        WHEN 'green' THEN 'go'
        WHEN 'yellow' THEN 'caution'
        WHEN 'red' THEN 'stop'
        ELSE 'you are not at a stop light' END as instructions
    FROM 'topic/subtopic'
```

The resulting output payload would be:

```
{
    "instructions":"caution"
}
```

Note

If *v* is `Undefined`, the result of the case statement is `Undefined`.

JSON extensions

You can use the following extensions to ANSI SQL syntax to facilitate work with nested JSON objects.

"." Operator

This operator accesses members in embedded JSON objects and functions identically to ANSI SQL and JavaScript. For example:

```
SELECT foo.bar AS bar.baz FROM 'topic/subtopic'
```

selects the value of the `bar` property in the `foo` object from the following message payload sent to the `topic/subtopic` topic.

```
{
  "foo": {
    "bar": "RED",
    "bar1": "GREEN",
    "bar2": "BLUE"
  }
}
```

If a JSON property name includes a hyphen character or numeric characters, the 'dot' notation will not work. Instead, you must use the get function to extract the property's value.

In this example, the following message is sent to the `iot/rules` topic.

```
{
  "mydata": {
    "item2": {
      "0": {
        "my-key": "myValue"
      }
    }
  }
}
```

Normally, the value of `my-key` would be identified as in this query.

```
SELECT * from iot/rules WHERE mydata.item2.0.my-key= "myValue"
```

However, because the property name `my-key` contains a hyphen and `item2` contains a numeric character, the get function must be used as the following query shows.

```
SELECT * from 'iot/rules' WHERE get(get(get(mydata,"item2"),"0"),"my-key") = "myValue"
```

* Operator

This functions in the same way as the * wildcard in ANSI SQL. It's used in the SELECT clause only and creates a new JSON object containing the message data. If the message payload is not in JSON format, * returns the entire message payload as raw bytes. For example:

```
SELECT * FROM 'topic/subtopic'
```

Applying a Function to an Attribute Value

The following is an example JSON payload that might be published by a device:

```
{
    "deviceid" : "iot123",
    "temp" : 54.98,
    "humidity" : 32.43,
    "coords" : {
        "latitude" : 47.615694,
        "longitude" : -122.3359976
    }
}
```

The following example applies a function to an attribute value in a JSON payload:

```
SELECT temp, md5(deviceid) AS hashed_id FROM topic/#
```

The result of this query is the following JSON object:

```
{
   "temp": 54.98,
   "hashed_id": "e37f81fb397e595c4aeb5645b8cbbbd1"
}
```

Substitution templates

You can use a substitution template to augment the JSON data returned when a rule is triggered and AWS IoT performs an action. The syntax for a substitution template is ${*expression*}, where *expression* can be any expression supported by AWS IoT in SELECT clauses, WHERE clauses, and AWS IoT rule actions. This expression can be plugged into an action field on a rule, allowing you to dynamically configure an action. In effect, this feature substitutes a piece of information in an action. This includes functions, operators, and information present in the original message payload.

> **Important**
>
> Because an expression in a substitution template is evaluated separately from the "SELECT ..." statement, you can't reference an alias created using the AS clause. You can only reference information present in the original payload, functions, and operators.

For more information about supported expressions, see AWS IoT SQL reference.

The following rule actions support substitution templates. Each action supports different fields that can be substituted.

- Apache Kafka
- CloudWatch alarms
- CloudWatch Logs
- CloudWatch metrics
- DynamoDB
- DynamoDBv2
- Elasticsearch
- HTTP
- AWS IoT Events
- AWS IoT SiteWise
- Kinesis Data Streams
- Firehose
- Lambda
- Location
- OpenSearch
- Republish

- S3
- SNS
- SQS
- Step Functions
- Timestream

Substitution templates appear in the action parameters within a rule:

```
{
    "sql": "SELECT *, timestamp() AS timestamp FROM 'my/iot/topic'",
    "ruleDisabled": false,
    "actions": [{
        "republish": {
            "topic": "${topic()}/republish",
            "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
        }
    }]
}
```

If this rule is triggered by the following JSON published to `my/iot/topic`:

```
{
    "deviceid": "iot123",
    "temp": 54.98,
    "humidity": 32.43,
    "coords": {
        "latitude": 47.615694,
        "longitude": -122.3359976
    }
}
```

Then this rule publishes the following JSON to `my/iot/topic/republish`, which AWS IoT substitutes from `${topic()}/republish`:

```
{
    "deviceid": "iot123",
    "temp": 54.98,
    "humidity": 32.43,
    "coords": {
```

```
        "latitude": 47.615694,
        "longitude": -122.3359976
    },
    "timestamp": 1579637878451
}
```

Nested object queries

You can use nested SELECT clauses to query for attributes within arrays and inner JSON objects. Supported by SQL version 2016-03-23 and later.

Consider the following MQTT message:

```
{
    "e": [
        { "n": "temperature", "u": "Cel", "t": 1234, "v": 22.5 },
        { "n": "light", "u": "lm", "t": 1235, "v": 135 },
        { "n": "acidity", "u": "pH", "t": 1235, "v": 7 }
    ]
}
```

Example

You can convert values to a new array with the following rule.

```
SELECT (SELECT VALUE n FROM e) as sensors FROM 'my/topic'
```

The rule generates the following output.

```
{
    "sensors": [
        "temperature",
        "light",
        "acidity"
    ]
}
```

Example

Using the same MQTT message, you can also query a specific value within a nested object with the following rule.

```
SELECT (SELECT v FROM e WHERE n = 'temperature') as temperature FROM 'my/topic'
```

The rule generates the following output.

```
{
    "temperature": [
        {
            "v": 22.5
        }
    ]
}
```

Example

You can also flatten the output with a more complicated rule.

```
SELECT get((SELECT v FROM e WHERE n = 'temperature'), 0).v as temperature FROM 'topic'
```

The rule generates the following output.

```
{
    "temperature": 22.5
}
```

Working with binary payloads

To handle your message payload as raw binary data (rather than a JSON object), you can use the * operator to refer to it in a SELECT clause.

In this topic:

- Binary payload examples
- Decoding protobuf message payloads

Binary payload examples

When you use * to refer to the message payload as raw binary data, you can add data to the rule. If you have an empty or a JSON payload, the resulting payload can have data added using the rule. The following shows examples of supported SELECT clauses.

- You can use the following SELECT clauses with only a * for binary payloads.
  - ```
    SELECT * FROM 'topic/subtopic'
    ```
  - ```
    SELECT * FROM 'topic/subtopic' WHERE timestamp() % 12 = 0
    ```
- You can also add data and use the following SELECT clauses.
  - ```
    SELECT *, principal() as principal, timestamp() as time FROM 'topic/subtopic'
    ```
  - ```
    SELECT encode(*, 'base64') AS data, timestamp() AS ts FROM 'topic/subtopic'
    ```
- You can also use these SELECT clauses with binary payloads.
  - The following refers to device_type in the WHERE clause.

    ```
    SELECT * FROM 'topic/subtopic' WHERE device_type = 'thermostat'
    ```
  - The following is also supported.

    ```
    {
     "sql": "SELECT * FROM 'topic/subtopic'",
     "actions": [
      {
       "republish": {
        "topic": "device/${device_id}"
       }
      }
     ]
    }
    ```

The following rule actions don't support binary payloads so you must decode them.

- Some rule actions don't support binary payload input, such as a Lambda action, so you must decode binary payloads. The Lambda rule action can receive binary data, if it's base64 encoded and in a JSON payload. You can do this by changing the rule to the following.

  ```
  SELECT encode(*, 'base64') AS data FROM 'my_topic'
  ```
- The SQL statement doesn't support string as input. To convert a string input to JSON, you can run the following command.

```
SELECT decode(encode(*, 'base64'), 'base64') AS payload FROM 'topic'
```

Decoding protobuf message payloads

Protocol Buffers (protobuf) is an open-source data format used to serialize structured data in a compact, binary form. It's used for transmitting data over networks or storing it in files. Protobuf allows you to send data in small packet sizes and at a faster rate than other messaging formats. AWS IoT Core Rules support protobuf by providing the decode(value, decodingScheme) SQL function, which allows you to decode protobuf-encoded message payloads to JSON format and route them to downstream services. This section details the step-by-step process to configure protobuf decoding in AWS IoT Core Rules.

In this section:

Prerequisites

- A basic understanding of Protocol Buffers (protobuf)
- The `.proto` files that define message types and related dependencies
- Installing Protobuf Compiler (protoc) on your system

Create descriptor files

If you already have your descriptor files, you can skip this step. A descriptor file (`.desc`) is a compiled version of a `.proto` file, which is a text file that defines the data structures and message types to be used in a protobuf serialization. To generate a descriptor file, you must define a `.proto` file and use the protoc compiler to compile it.

1. Create `.proto` files that define the message types. An example `.proto` file can look like the following:

```
syntax = "proto3";

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

   In this example `.proto` file, you use proto3 syntax and define message type `Person`. The `Person` message definition specifies three fields (name, id, and email). For more information about `.proto` file message formats, see Language Guide (proto3).

2. Use the protoc compiler to compile the `.proto` files and generate a descriptor file. An example command to create a descriptor (`.desc`) file can be the following:

```
protoc --descriptor_set_out=<FILENAME>.desc \
    --proto_path=<PATH_TO_IMPORTS_DIRECTORY> \
    --include_imports \
    <PROTO_FILENAME>.proto
```

   This example command generates a descriptor file `<FILENAME>.desc`, which AWS IoT Core Rules can use to decode protobuf payloads that conform to the data structure defined in `<PROTO_FILENAME>.proto`.

   - `--descriptor_set_out`

     Specifies the name of the descriptor file (`<FILENAME>.desc`) that should be generated.

   - `--proto_path`

     Specifies the locations of any imported `.proto` files that are referenced by the file being compiled. You can specify the flag multiple times if you have multiple imported `.proto` files with different locations.

   - `--include_imports`

     Specifies that any imported `.proto` files should also be compiled and included in the `<FILENAME>.desc` descriptor file.

- <PROTO_FILENAME>.proto

  Specifies the name of the .proto file that you want to compile.

For more information about the protoc reference, see API Reference.

Upload descriptor files to S3 bucket

After you create your descriptor files <FILENAME>.desc, upload the descriptor files <FILENAME>.desc to an Amazon S3 bucket, using the AWS API, AWS SDK, or the AWS Management Console.

Important considerations

- Make sure that you upload the descriptor files to an Amazon S3 bucket in your AWS account in the same AWS Region where you intend to configure your Rules.
- Make sure that you grant AWS IoT Core access to read the FileDescriptorSet from S3. If your S3 bucket has server-side encryption (SSE) disabled or if your S3 bucket is encrypted using Amazon S3-managed keys (SSE-S3), no additional policy configurations are required. This can be accomplished with the example bucket policy:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Sid": "Statement1",
   "Effect": "Allow",
   "Principal": {
    "Service": "iot.amazonaws.com"
   },
   "Action": "s3:Get*",
                       "Resource": "arn:aws:s3:::<BUCKET NAME>/<FILENAME>.desc"
  }
 ]
}
```

- If your S3 bucket is encrypted using an AWS Key Management Service key (SSE-KMS), make sure that you grant AWS IoT Core permission to use the key when accessing your S3 bucket. You can do this by adding this statement to your key policy:

```
{
 "Sid": "Statement1",
 "Effect": "Allow",
 "Principal": {
  "Service": "iot.amazonaws.com"
 },
 "Action": [
  "kms:Decrypt",
  "kms:GenerateDataKey*",
  "kms:DescribeKey"
 ],
        "Resource": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"

}
```

Configure protobuf decoding in Rules

After you upload the descriptor files to your Amazon S3 bucket, configure a Rule that can decode your protobuf message payload format using the decode(value, decodingScheme) SQL function. A detailed function signature and example can be found in the decode(value, decodingScheme) SQL function of the *AWS IoT SQL reference*.

The following is an example SQL expression using the decode(value, decodingScheme) function:

```
SELECT VALUE decode(*, 'proto', '<BUCKET NAME>', '<FILENAME>.desc', '<PROTO_FILENAME>',
 '<PROTO_MESSAGE_TYPE>') FROM '<MY_TOPIC>'
```

In this example expression:

- You use the decode(value, decodingScheme) SQL function to decode the binary message payload referenced by `*`. This can be a binary protobuf-encoded payload or a JSON string that represents a base64-encoded protobuf payload.
- The message payload provided is encoded using the `Person` message type defined in `PROTO_FILENAME.proto`.
- The Amazon S3 bucket named `BUCKET NAME` contains the `FILENAME.desc` generated from `PROTO_FILENAME.proto`.

After you complete the configuration, publish a message to AWS IoT Core on the topic to which the Rule is subscribed.

Limitations

AWS IoT Core Rules support protobuf with the following limitations:

- Decoding protobuf message payloads within substitution templates is not supported.
- When decoding protobuf message payloads, you can use the decode SQL function within a single SQL expression up to two times.
- The maximum inbound payload size is 128 KiB (1KiB =1024 bytes), the maximum outbound payload size is 128 KiB, and the maximum size for a `FileDescriptorSet` object stored in an Amazon S3 bucket is 32 KiB.
- Amazon S3 buckets encrypted with SSE-C encryption are not supported.

Best practices

Here are some best practices and troubleshooting tips.

- Back up your proto files in the Amazon S3 bucket.

  It's a good practice to back up your proto files in case something goes wrong. For example, if you incorrectly modify the proto files without backups when running protoc, this can cause issues in your production stack. There are multiple ways to back up your files in an Amazon S3 bucket. For example, you can use versioning in S3 buckets. For more information about how to back up files in Amazon S3 buckets, refer to the *Amazon S3 Developer Guide*.
- Configure AWS IoT logging to view log entries.

  It's a good practice to configure AWS IoT logging so that you can check AWS IoT logs for your account in CloudWatch. When a rule's SQL query calls an external function, AWS IoT Core Rules generates a log entry with an `eventType` of `FunctionExecution`, which contains the reason field that will help you troubleshoot failures. Possible errors include an Amazon S3 object not found, or invalid protobuf file descriptor. For more information about how to configure AWS IoT logging and see the log entries, see Configure AWS IoT logging and Rules engine log entries.
- Update `FileDescriptorSet` using a new object key and update the object key in your Rule.

  You can update `FileDescriptorSet` by uploading an updated descriptor file to your Amazon S3 bucket. Your updates to `FileDescriptorSet` can take up to 15 minutes to be reflected. To avoid this delay, it's a good practice to upload your updated `FileDescriptorSet` using a new object key, and update the object key in your Rule.

SQL versions

The AWS IoT rules engine uses an SQL-like syntax to select data from MQTT messages. The SQL statements are interpreted based on an SQL version specified with the `awsIotSqlVersion` property in a JSON document that describes the rule. For more information about the structure of JSON rule documents, see Creating a Rule. The `awsIotSqlVersion` property lets you specify which version of the AWS IoT SQL rules engine that you want to use. When a new version is deployed, you can continue to use an earlier version or change your rule to use the new version. Your current rules continue to use the version with which they were created.

The following JSON example shows you how to specify the SQL version using the `awsIotSqlVersion` property.

```
{
    "sql": "expression",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [{
        "republish": {
            "topic": "my-mqtt-topic",
            "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
        }
    }]
}
```

AWS IoT currently supports the following SQL versions:

- `2016-03-23` – The SQL version built on 2016-03-23 (recommended).
- `2015-10-08` – The original SQL version built on 2015-10-08.
- `beta` – The most recent beta SQL version. This version could introduce breaking changes to your rules.

What's new in the 2016-03-23 SQL rules engine version

- Fixes for selecting nested JSON objects.

- Fixes for array queries.
- Intra-object query support. For more information, see Nested object queries.
- Support to output an array as a top-level object.
- Addition of the `encode(value, encodingScheme)` function, which can be applied on JSON and non-JSON format data. For more information, see the encode function.

Output an `Array` as a top-level object

This feature allows a rule to return an array as a top-level object. For example, given the following MQTT message:

```
{
    "a": {"b":"c"},
    "arr":[1,2,3,4]
}
```

And the following rule:

```
SELECT VALUE arr FROM 'topic'
```

The rule generates the following output.

```
[1,2,3,4]
```

AWS IoT Device Shadow service

The AWS IoT Device Shadow service adds shadows to AWS IoT thing objects. Shadows can make a device’s state available to apps and other services whether the device is connected to AWS IoT or not. AWS IoT thing objects can have multiple named shadows so that your IoT solution has more options for connecting your devices to other apps and services.

AWS IoT thing objects don't have any shadows until they are created explicitly. Shadows can be created, updated, and deleted by using the AWS IoT console. Devices, other web clients, and services can create, update, and delete shadows by using MQTT and the reserved MQTT topics, HTTP using the Device Shadow REST API, and the AWS CLI for AWS IoT. Because shadows are stored by AWS in the cloud, they can collect and report device state data from apps and other cloud services whether the device is connected or not.

Using shadows

Shadows provide a reliable data store for devices, apps, and other cloud services to share data. They enable devices, apps, and other cloud services to connect and disconnect without losing a device's state.

While devices, apps, and other cloud services are connected to AWS IoT, they can access and control the current state of a device through its shadows. For example, an app can request a change in a device's state by updating a shadow. AWS IoT publishes a message that indicates the change to the device. The device receives this message, updates its state to match, and publishes a message with its updated state. The Device Shadow service reflects this updated state in the corresponding shadow. The app can subscribe to the shadow's update or it can query the shadow for its current state.

When a device goes offline, an app can still communicate with AWS IoT and the device's shadows. When the device reconnects, it receives the current state of its shadows so that it can update its state to match that of its shadows, and then publish a message with its updated state. Likewise, when an app goes offline and the device state changes while it's offline, the device keeps the shadow updated so the app can query the shadows for its current state when it reconnects.

If your devices are frequently offline and you would like to configure your devices to receive delta messages after they reconnect, you can use the persistent session feature. For more information about the persistent session expiry period, see Persistent session expiry period.

Choosing to use named or unnamed shadows

The Device Shadow service supports named and unnamed, or classic, shadows. A thing object can have multiple named shadows, and no more than one unnamed shadow. The thing object can also have a reserved named shadow, which operates similarly to a named shadow except that you can't update its name. For more information, see Reserved named shadow.

A thing object can have both named and unnamed shadows at the same time; however, the API used to access each is slightly different, so it might be more efficient to decide which type of shadow would work best for your solution and use that type only. For more information about the API to access the shadows, see Shadow topics.

With named shadows, you can create different views of a thing object's state. For example, you could divide a thing object with many properties into shadows with logical groups of properties, each identified by its shadow name. You could also limit access to properties by grouping them into different shadows and using policies to control access. For more information about policies to use with device shadows, see Actions, resources, and condition keys for AWS IoT and AWS IoT Core policies.

The classic, unnamed shadows are simpler, but somewhat more limited than the named shadows. Each AWS IoT thing object can have only one unnamed shadow. If you expect your IoT solution to have a limited need for shadow data, this might be how you want to get started using shadows. However, if you think you might want to add additional shadows in the future, consider using named shadows from the start.

Fleet indexing supports unnamed shadows and named shadows differently. For more information, see Manage fleet indexing.

Accessing shadows

Every shadow has a reserved MQTT topic and HTTP URL that supports the `get`, `update`, and `delete` actions on the shadow.

Shadows use JSON shadow documents to store and retrieve data. A shadow's document contains a state property that describes these aspects of the device's state:

- `desired`

  Apps specify the desired states of device properties by updating the `desired` object.

- `reported`

  Devices report their current state in the `reported` object.
- `delta`

  AWS IoT reports differences between the desired and the reported state in the `delta` object.

The data stored in a shadow is determined by the state property of the update action's message body. Subsequent update actions can modify the values of an existing data object, and also add and delete keys and other elements in the shadow’s state object. For more information about accessing shadows, see Using shadows in devices and Using shadows in apps and services.

> **Important**
>
> Permission to make update requests should be limited to trusted apps and devices. This prevents the shadow's state property from being changed unexpectedly; otherwise, the devices and apps that use the shadow should be designed to expect the keys in the state property to change.

Using shadows in devices, apps, and other cloud services

Using shadows in devices, apps, and other cloud services requires consistency and coordination between all of these. The AWS IoT Device Shadow service stores the shadow state, sends messages when the shadow state changes, and responds to messages that change its state. The devices, apps, and other cloud services in your IoT solution must manage their state and keep it consistent with the device shadow's state.

The shadow state data is dynamic and can be altered by the devices, apps, and other cloud services with permission to access the shadow. For this reason, it is important to consider how each device, app, and other cloud service will interact with the shadow. For example:

- *Devices* should write only to the `reported` property of the shadow state when communicating state data to the shadow.
- *Apps and other cloud services* should write only to the `desired` property when communicating state change requests to the device through the shadow.

> **Important**
>
> The data contained in a shadow data object is independent from that of other shadows and other thing object properties, such as a thing's attributes and the content of MQTT messages that a thing object's device might publish. A device can, however, report the same data in different MQTT topics and shadows if necessary.
>
> A device that supports multiple shadows must maintain the consistency of the data that it reports in the different shadows.

Message order

There is no guarantee that messages from the AWS IoT service will arrive at the device in any specific order. The following scenario shows what happens in this case.

Initial state document:

```
{
  "state": {
    "reported": {
      "color": "blue"
    }
  },
  "version": 9,
  "timestamp": 123456776
}
```

Update 1:

```
{
  "state": {
    "desired": {
      "color": "RED"
    }
  },
  "version": 10,
  "timestamp": 123456777
}
```

Update 2:

```
{
  "state": {
    "desired": {
      "color": "GREEN"
    }
  },
  "version": 11,
  "timestamp": 123456778
}
```

Final state document:

```
{
  "state": {
    "reported": {
      "color": "GREEN"
    }
  },
  "version": 12,
  "timestamp": 123456779
}
```

This results in two delta messages:

```
{
  "state": {
    "color": "RED"
  },
  "version": 11,
  "timestamp": 123456778
}
```

```
{
  "state": {
    "color": "GREEN"
  },
  "version": 12,
  "timestamp": 123456779
}
```

The device might receive these messages out of order. Because the state in these messages is cumulative, a device can safely discard any messages that contain a version number older than the one it is tracking. If the device receives the delta for version 12 before version 11, it can safely discard the version 11 message.

Trim shadow messages

To reduce the size of shadow messages sent to your device, define a rule that selects only the fields your device needs then republishes the message on an MQTT topic to which your device is listening.

The rule is specified in JSON and should look like the following:

```
{
  "sql": "SELECT state, version FROM '$aws/things/+/shadow/update/delta'",
  "ruleDisabled": false,
  "actions": [
    {
      "republish": {
        "topic": "${topic(3)}/delta",
        "roleArn": "arn:aws:iam:123456789012:role/my-iot-role"
      }
    }
  ]
}
```

The SELECT statement determines which fields from the message will be republished to the specified topic. A "+" wild card is used to match all shadow names. The rule specifies that all matching messages should be republished to the specified topic. In this case, the `"topic()"` function is used to specify the topic on which to republish. `topic(3)` evaluates to the thing name in the original topic. For more information about creating rules, see Rules for AWS IoT.

Using shadows in devices

This section describes device communications with shadows using MQTT messages, the preferred method for devices to communicate with the AWS IoT Device Shadow service.

Shadow communications emulate a request/response model using the publish/subscribe communication model of MQTT. Every shadow action consists of a request topic, a successful response topic (`accepted`), and an error response topic (`rejected`).

If you want apps and services to be able to determine whether a device is connected, see Detecting if a device is connected.

> **Important**
>
> Because MQTT uses a publish/subscribe communication model, you should subscribe to the response topics *before* you publish a request topic. If you don't, you might not receive the response to the request that you publish.
> If you use an AWS IoT Device SDK to call the Device Shadow service APIs, this is handled for you.

The examples in this section use an abbreviated form of the topic where the *ShadowTopicPrefix* can refer to either a named or an unnamed shadow, as described in this table.

Shadows can be named or unnamed (classic). The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

ShadowTopicPrefix **value**	**Shadow type**
`$aws/things/` *thingName* `/shadow`	Unnamed (classic) shadow
`$aws/things/` *thingName* `/shadow/name/` *shadowName*	Named shadow

> **Important**
>
> Make sure that your app's or service's use of the shadows is consistent and supported by the corresponding implementations in your devices. For example, consider how shadows are created, updated, and deleted. Also consider how updates are handled in the device and the apps or services that access the device through a shadow. Your design should be clear about how the device's state is updated and reported and how your apps and services interact with the device and its shadows.

To create a complete topic, select the *ShadowTopicPrefix* for the type of shadow to which you want to refer, replace *thingName*, and *shadowName* if applicable, with their corresponding values, and then append that with the topic stub as shown in the following table. Remember that topics are case sensitive.

See Shadow topics for more information about the reserved topics for shadows.

Initializing the device on first connection to AWS IoT

After a device registers with AWS IoT, it should subscribe to these MQTT messages for the shadows that it supports.

Topic	Meaning	Action a device should take when this topic is received
ShadowTopicPrefix / delete/accepted	The `delete` request was accepted and AWS IoT deleted the shadow.	The actions necessary to accommodate the deleted shadow, such as stop publishing updates.
ShadowTopicPrefix / delete/rejected	The `delete` request was rejected by AWS IoT and the shadow was not deleted. The message body contains the error information.	Respond to the error message in the message body.
ShadowTopicPrefix / get/accepted	The `get` request was accepted by AWS IoT, and the message body contains the current shadow document.	The actions necessary to process the state document in the message body.
ShadowTopicPrefix / get/rejected	The `get` request was rejected by AWS IoT, and the message body contains the error information.	Respond to the error message in the message body.
ShadowTopicPrefix / update/accepted	The `update` request was accepted by AWS IoT, and the message body contains the current shadow document.	Confirm the updated data in the message body matches the device state.

Topic	Meaning	Action a device should take when this topic is received
ShadowTopicPrefix / update/rejected	The `update` request was rejected by AWS IoT, and the message body contains the error information.	Respond to the error message in the message body.
ShadowTopicPrefix / update/delta	The shadow document was updated by a request to AWS IoT, and the message body contains the changes requested.	Update the device's state to match the desired state in the message body.
ShadowTopicPrefix / update/documents	An update to the shadow was recently completed, and the message body contains the current shadow document.	Confirm the updated state in the message body matches the device's state.

After subscribing to the messages in the preceding table for each shadow, the device should test to see if the shadows that it supports have already been created by publishing a `/get` topic to each shadow. If a `/get/accepted` message is received, the message body contains the shadow document, which the device can use to initialize its state. If a `/get/rejected` message is received, the shadow should be created by publishing an `/update` message with the current device state.

For example, suppose you have a thing `My_IoT_Thing` which doesn't have any classic or named shadows. If you now publish a `/get` request on the reserved topic `$aws/things/My_IoT_Thing/shadow/get`, it returns an error on the `$aws/things/My_IoT_Thing/shadow/get/rejected` topic because the thing doesn't have any shadows. To resolve this error, first publish an `/update` message by using the `$aws/things/My_IoT_Thing/shadow/update` topic with the current device state such as the following payload.

```
{
 "state": {
  "reported": {
   "welcome": "aws-iot",
   "color": "yellow"
  }
```

```
  }
}
```

A classic shadow is now created for the thing and the message is published to the `$aws/things/My_IoT_Thing/shadow/update/accepted` topic. If you publish to the topic `$aws/things/My_IoT_Thing/shadow/get`, it returns a response to the `$aws/things/My_IoT_Thing/shadow/get/accepted` topic with the device state.

For named shadows, you must first create the named shadow or publish an update with the shadow name before using the get request. For example, to create a named shadow `namedShadow1`, first publish the device state information to the topic `$aws/things/My_IoT_Thing/shadow/name/namedShadow1/update`. To retrieve the state information, use the `/get` request for the named shadow, `$aws/things/My_IoT_Thing/shadow/name/namedShadow1/get`.

Processing messages while the device is connected to AWS IoT

While a device is connected to AWS IoT, it can receive **/update/delta** messages and should keep the device state matched to the changes in its shadows by:

1. Reading all **/update/delta** messages received and synchronizing the device state to match.
2. Publishing an **/update** message with a `reported` message body that has the device's current state, whenever the device's state changes.

While a device is connected, it should publish these messages when indicated.

Indication	Topic	Payload
The device's state has changed.	*ShadowTopicPrefix* /update	A shadow document with the `reported` property.
The device might not be synchronized with the shadow.	*ShadowTopicPrefix* /get	(empty)
An action on the device indicates that a shadow will no longer be supported by	*ShadowTopicPrefix* /delete	(empty)

Indication	Topic	Payload
the device, such as when the device is being removed or replaced.		

Processing messages when the device reconnects to AWS IoT

When a device with one or more shadows connects to AWS IoT, it should synchronize its state with that of all the shadows that it supports by:

1. Reading all **/update/delta** messages received and synchronizing the device state to match.
2. Publishing an **/update** message with a `reported` message body that has the device's current state.

Using shadows in apps and services

This section describes how an app or service interacts with the AWS IoT Device Shadow service. This example assumes the app or service is interacting only with the shadow and, through the shadow, the device. This example doesn't include any management actions, such as creating or deleting shadows.

This example uses the AWS IoT Device Shadow service's REST API to interact with shadows. Unlike the example used in Using shadows in devices, which uses a publish/subscribe communications model, this example uses the request/response communications model of the REST API. This means the app or service must make a request before it can receive a response from AWS IoT. A disadvantage of this model, however, is that it does not support notifications. If your app or service requires timely notifications of device state changes, consider the MQTT or MQTT over WSS protocols, which support the publish/subscribe communication model, as described in Using shadows in devices.

> **Important**
>
> Make sure that your app's or service's use of the shadows is consistent with and supported by the corresponding implementations in your devices. Consider, for example, how shadows are created, updated, and deleted, and how updates are handled in the device and the apps or services that access the shadow. Your design should clearly specify how the device's state

is updated and reported, and how your apps and services interact with the device and its shadows.

The REST API's URL for a named shadows is:

```
https://endpoint/things/thingName/shadow?name=shadowName
```

and for an unnamed shadow:

```
https://endpoint/things/thingName/shadow
```

where:

endpoint

The endpoint returned by the CLI command:

```
aws iot describe-endpoint --endpoint-type IOT:Data-ATS
```

thingName

The name of the thing object to which the shadow belongs

shadowName

The name of the named shadow. This parameter is not used with unnamed shadows.

Initializing the app or service on connection to AWS IoT

When the app first connects to AWS IoT, it should send an HTTP GET request to the URLs of the shadows it uses to get the current state of the shadows it's using. This allows it to sync the app or service to the shadow.

Processing state changes while the app or service is connected to AWS IoT

While the app or service is connected to AWS IoT, it can query the current state periodically by sending an HTTP GET request on the URLs of the shadows it uses.

When an end user interacts with the app or service to change the state of the device, the app or service can send an HTTP POST request to the URLs of the shadows it uses to update the `desired` state of the shadow. This request returns the change that was accepted, but you might have to poll the shadow by making HTTP GET requests until the device has updated the shadow with its new state.

Detecting if a device is connected

To determine if a device is currently connected, include a `connected` property in the shadow document and use an MQTT Last Will and Testament (LWT) message to set the `connected` property to `false` if a device is disconnected due to an error.

> **Note**
>
> MQTT LWT messages sent to AWS IoT reserved topics (topics that begin with $) are ignored by the AWS IoT Device Shadow service. However, they are processed by subscribed clients and by the AWS IoT rules engine, so you will need to create an LWT message that is sent to a non-reserved topic and a rule that republishes the MQTT LWT message as a shadow update message to the shadow's reserved update topic, *`ShadowTopicPrefix`*`/update`.

To send the Device Shadow service an LWT message

1. Create a rule that republishes the MQTT LWT message on the reserved topic. The following example is a rule that listens for messages on the `my/things/myLightBulb/update` topic and republishes it to `$aws/things/myLightBulb/shadow/update`.

```
{
    "rule": {
    "ruleDisabled": false,
    "sql": "SELECT * FROM 'my/things/myLightBulb/update'",
    "description": "Turn my/things/ into $aws/things/",
    "actions": [
        {
        "republish": {
            "topic": "$$aws/things/myLightBulb/shadow/update",
            "roleArn": "arn:aws:iam:123456789012:role/aws_iot_republish"
            }
        }
     ]
```

```
    }
}
```

2. When the device connects to AWS IoT, it registers an LWT message to a non-reserved topic for the republish rule to recognize. In this example, that topic is `my/things/myLightBulb/update` and it sets the connected property to `false`.

```
{
    "state": {
        "reported": {
            "connected":"false"
        }
    }
}
```

3. After connecting, the device publishes a message on its shadow update topic, `$aws/things/myLightBulb/shadow/update`, to report its current state, which includes setting its `connected` property to `true`.

```
{
     "state": {
        "reported": {
            "connected":"true"
        }
    }
}
```

4. Before the device disconnects gracefully, it publishes a message on its shadow update topic, `$aws/things/myLightBulb/shadow/update`, to report its latest state, which include setting its `connected` property to `false`.

```
{
    "state": {
        "reported": {
            "connected":"false"
        }
    }
}
```

5. If the device disconnects due to an error, the AWS IoT message broker publishes the device's LWT message on behalf of the device. The republish rule detects this message and publishes the shadow update message to update the `connected` property of the device shadow.

Note

Due to the asynchronous nature of disconnect processing, LWT messages are not guaranteed to be dispatched in order during reconnection. We recommend that you use lifecycle events to improve the accuracy of connectivity state detection, as these events provide attributes to manage out-of-order events.

Simulating Device Shadow service communications

This topic demonstrates how the Device Shadow service acts as an intermediary and allows devices and apps to use a shadow to update, store, and retrieve a device's state.

To demonstrate the interaction described in this topic, and to explore it further, you'll need an AWS account and a system on which you can run the AWS CLI. If you don't have these, you can still see the interaction in the code examples.

In this example, the AWS IoT console represents the device. The AWS CLI represents the app or service that accesses the device by way of the shadow. The AWS CLI interface is very similar to the API that an app might use to communicate with AWS IoT. The device in this example is a smart light bulb and the app displays the light bulb's state and can change the light bulb's state.

Setting up the simulation

These procedures initialize the simulation by opening the AWS IoT console, which simulates your device, and the command line window that simulates your app.

To set up your simulation environment

1. You'll need an AWS account to run the examples from this topic on your own. If you don't have an AWS account, create one, as described in Set up AWS account.
2. Open the AWS IoT console, and in the left menu, choose **Test** to open the **MQTT client**.
3. In another window, open a terminal window on a system that has the AWS CLI installed on it.

You should have two windows open: one with the AWS IoT console on the **Test** page, and one with a command line prompt.

Initialize the device

In this simulation, we'll be working with a thing object named, *mySimulatedThing*, and its shadow named, *simShadow1*.

Create thing object and its IoT policy

To create a thing object, in the **AWS IoT Console**:

1. Choose **Manage** and then choose **Things**.
2. Click the **Create** button if things are listed otherwise click **Register a single thing** to create a single AWS IoT thing.
3. Enter the name `mySimulatedThing`, leave other settings to default, and then click **Next**.
4. Use one-click certificate creation to generate the certificates that will authenticate the device's connection to AWS IoT. Click **Activate** to activate the certificate.
5. You can attach the policy `My_IoT_Policy` that would give the device permission to publish and subscribe to the MQTT reserved topics. For more detailed steps about how to create an AWS IoT thing and how to create this policy, see Create a thing object.

Create named shadow for the thing object

You can create a named shadow for a thing by publishing an update request to the topic `$aws/things/mySimulatedThing/shadow/name/simShadow1/update` as described below.

Or, to create a named shadow:

1. In the **AWS IoT Console**, choose your thing object in the list of things displayed and then choose **Shadows**.
2. Choose **Add a shadow**, enter the name `simShadow1`, and then choose **Create** to add the named shadow.

Subscribe and publish to reserved MQTT topics

In the console, subscribe to the reserved MQTT shadow topics. These topics are the responses to the `get`, `update`, and `delete` actions so that your device will be ready to receive the responses after it publishes an action.

To subscribe to an MQTT topic in the MQTT client

1. In the **MQTT client**, choose **Subscribe to a topic**.
2. Enter the `get`, `update`, and `delete` topics to subscribe to. Copy one topic at a time from the following list, paste it in the **Topic filter** field, and then click **Subscribe**. You should see the topics appear under **Subscriptions**.

    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/delete/accepted`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/delete/rejected`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/get/accepted`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/get/rejected`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/rejected`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/delta`
    - `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/documents`

    At this point, your simulated device is ready to receive the topics as they are published by AWS IoT.

To publish to an MQTT topic in the MQTT client

After a device has initialized itself and subscribed to the response topics, it should query for the shadows it supports. This simulation supports only one shadow, the shadow that supports a thing object named, *mySimulatedThing*, named, *simShadow1*.

To get the current shadow state from the MQTT client

1. In the **MQTT client**, choose **Publish to a topic**.
2. Under **Publish**, enter the following topic and delete any content from the message body window below where you entered the topic to get. You can then choose **Publish to topic** to publish the request. `$aws/things/mySimulatedThing/shadow/name/simShadow1/get`.

    If you haven't created the named shadow, `simShadow1`, you receive a message in the `$aws/things/mySimulatedThing/shadow/name/simShadow1/get/rejected` topic and the `code` is `404`, such as in this example as the shadow has not been created, so we'll create it next.

```
{
  "code": 404,
  "message": "No shadow exists with name: 'simShadow1'"
}
```

To create a shadow with the current status of the device

1. In the **MQTT client**, choose **Publish to a topic** and enter this topic:

```
$aws/things/mySimulatedThing/shadow/name/simShadow1/update
```

2. In the message body window below where you entered the topic, enter this shadow document to show the device is reporting its ID and its current color in RGB values. Choose **Publish** to publish the request.

```
{
  "state": {
    "reported": {
      "ID": "SmartLamp21",
      "ColorRGB": [
        128,
        128,
        128
      ]
    }
  },
  "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"
}
```

If you receive a message in the topic:

- `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted`: It means that the shadow was created and the message body contains the current shadow document.
- `$aws/things/mySimulatedThing/shadow/name/simShadow1/update/rejected`: Review the error in the message body.

- `$aws/things/mySimulatedThing/shadow/name/simShadow1/get/accepted`: The shadow already exists and the message body has the current shadow state, such as in this example. With this, you could set your device or confirm that it matches the shadow state.

```
{
  "state": {
    "reported": {
      "ID": "SmartLamp21",
      "ColorRGB": [
        128,
        128,
        128
      ]
    }
  },
  "metadata": {
    "reported": {
      "ID": {
        "timestamp": 1591140517
      },
      "ColorRGB": [
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        }
      ]
    }
  },
  "version": 3,
  "timestamp": 1591140517,
  "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"
}
```

Send an update from the app

This section uses the AWS CLI to demonstrate how an app can interact with a shadow.

To get the current state of the shadow using the AWS CLI

From the command line, enter this command.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1 /
dev/stdout
```

On Windows platforms, you can use `con` instead of `/dev/stdout`.

```
aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name simShadow1
 con
```

Because the shadow exists and had been initialized by the device to reflect its current state, it should return the following shadow document.

```
{
  "state": {
    "reported": {
      "ID": "SmartLamp21",
      "ColorRGB": [
        128,
        128,
        128
      ]
    }
  },
  "metadata": {
    "reported": {
      "ID": {
        "timestamp": 1591140517
      },
      "ColorRGB": [
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        },
        {
          "timestamp": 1591140517
        }
      ]
```

```
    }
  },
  "version": 3,
  "timestamp": 1591141111
}
```

The app can use this response to initialize its representation of the device state.

If the app updates the state, such as when an end user changes the color of our smart light bulb to yellow, the app would send an **update-thing-shadow** command. This command corresponds to the `UpdateThingShadow` REST API.

To update a shadow from an app

From the command line, enter this command.

AWS CLI v2.x

```
aws iot-data update-thing-shadow --thing-name mySimulatedThing --shadow-name
 simShadow1 \
    --cli-binary-format raw-in-base64-out \
    --payload '{"state":{"desired":{"ColorRGB":
[255,255,0]}},"clientToken":"21b21b21-bfd2-4279-8c65-e2f697ff4fab"}' /dev/stdout
```

AWS CLI v1.x

```
aws iot-data update-thing-shadow --thing-name mySimulatedThing --shadow-name
 simShadow1 \
    --payload '{"state":{"desired":{"ColorRGB":
[255,255,0]}},"clientToken":"21b21b21-bfd2-4279-8c65-e2f697ff4fab"}' /dev/stdout
```

If successful, this command should return the following shadow document.

```
{
  "state": {
    "desired": {
      "ColorRGB": [
        255,
        255,
        0
      ]
```

```
    }
  },
  "metadata": {
    "desired": {
      "ColorRGB": [
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        },
        {
          "timestamp": 1591141596
        }
      ]
    }
  },
  "version": 4,
  "timestamp": 1591141596,
  "clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"
}
```

Respond to update in device

Returning to the **MQTT client** in the AWS console, you should see the messages that AWS IoT published to reflect the update command issued in the previous section.

To view the update messages in the MQTT client

In the **MQTT client**, choose **$aws/things/mySimulatedThing/shadow/name/simShadow1/update/delta** in the **Subscriptions** column. If the topic name is truncated, you can pause on it to see the full topic. In the topic log of this topic, you should see a `/delta` message similar to this one.

```
{
  "version": 4,
  "timestamp": 1591141596,
  "state": {
    "ColorRGB": [
      255,
      255,
      0
    ]
```

```
  },
  "metadata": {
    "ColorRGB": [
      {
        "timestamp": 1591141596
      },
      {
        "timestamp": 1591141596
      },
      {
        "timestamp": 1591141596
      }
    ]
  },
  "clientToken": "21b21b21-bfd2-4279-8c65-e2f697ff4fab"
}
```

Your device would process the contents of this message to set the device state to match the `desired` state in the message.

After the device updates the state to match the `desired` state in the message, it must send the new reported state back to AWS IoT by publishing an update message. This procedure simulates this in the **MQTT client**.

To update the shadow from the device

1. In the **MQTT client**, choose **Publish to a topic**.
2. In the message body window , in the topic field above the message body window, enter the shadow's topic followed by the `/update` action: `$aws/things/mySimulatedThing/shadow/name/simShadow1/update` and in the message body, enter this updated shadow document, which describes the current state of the device. Click **Publish** to publish the updated device state.

```
{
  "state": {
    "reported": {
      "ColorRGB": [255,255,0]
      }
  },
  "clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}
```

If the message was successfully received by AWS IoT, you should see a new response in the **$aws/things/mySimulatedThing/shadow/name/simShadow1/update/accepted** message log in the **MQTT client** with the current state of the shadow, such as this example.


```
{
  "state": {
    "reported": {
      "ColorRGB": [
        255,
        255,
        0
      ]
    }
  },
  "metadata": {
    "reported": {
      "ColorRGB": [
        {
          "timestamp": 1591142747
        },
        {
          "timestamp": 1591142747
        },
        {
          "timestamp": 1591142747
        }
      ]
    }
  },
  "version": 5,
  "timestamp": 1591142747,
  "clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}
```


A successful update to the reported state of the device also causes AWS IoT to send a comprehensive description of the shadow state in a message to the `update/documents` topic, such as this message body that resulted from the shadow update performed by the device in the preceding procedure.

```
{
```

```
  "previous": {
    "state": {
      "desired": {
        "ColorRGB": [
          255,
          255,
          0
        ]
      },
      "reported": {
        "ID": "SmartLamp21",
        "ColorRGB": [
          128,
          128,
          128
        ]
      }
    },
    "metadata": {
      "desired": {
        "ColorRGB": [
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          }
        ]
      },
      "reported": {
        "ID": {
          "timestamp": 1591140517
        },
        "ColorRGB": [
          {
            "timestamp": 1591140517
          },
          {
            "timestamp": 1591140517
          },
          {
```

```
            "timestamp": 1591140517
          }
        ]
      }
    },
    "version": 4
  },
  "current": {
    "state": {
      "desired": {
        "ColorRGB": [
          255,
          255,
          0
        ]
      },
      "reported": {
        "ID": "SmartLamp21",
        "ColorRGB": [
          255,
          255,
          0
        ]
      }
    },
    "metadata": {
      "desired": {
        "ColorRGB": [
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          }
        ]
      },
      "reported": {
        "ID": {
          "timestamp": 1591140517
        },
        "ColorRGB": [
```

```
          {
            "timestamp": 1591142747
          },
          {
            "timestamp": 1591142747
          },
          {
            "timestamp": 1591142747
          }
        ]
      }
    },
    "version": 5
  },
  "timestamp": 1591142747,
  "clientToken": "a4dc2227-9213-4c6a-a6a5-053304f60258"
}
```


Observe the update in the app

The app can now query the shadow for the current state as reported by the device.

To get the current state of the shadow using the AWS CLI

1. From the command line, enter this command.

   ```
   aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name
    simShadow1 /dev/stdout
   ```

   On Windows platforms, you can use `con` instead of `/dev/stdout`.

   ```
   aws iot-data get-thing-shadow --thing-name mySimulatedThing --shadow-name
    simShadow1 con
   ```

2. Because the shadow has just been updated by the device to reflect its current state, it should return the following shadow document.


   ```
   {
     "state": {
       "desired": {
         "ColorRGB": [
           255,
   ```

```
          255,
          0
        ]
      },
      "reported": {
        "ID": "SmartLamp21",
        "ColorRGB": [
          255,
          255,
          0
        ]
      }
    },
    "metadata": {
      "desired": {
        "ColorRGB": [
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          },
          {
            "timestamp": 1591141596
          }
        ]
      },
      "reported": {
        "ID": {
          "timestamp": 1591140517
        },
        "ColorRGB": [
          {
            "timestamp": 1591142747
          },
          {
            "timestamp": 1591142747
          },
          {
            "timestamp": 1591142747
          }
        ]
      }
    },
```

```
  "version": 5,
  "timestamp": 1591143269
}
```

Going beyond the simulation

Experiment with the interaction between the AWS CLI (representing the app) and the console (representing the device) to model your IoT solution.

Interacting with shadows

This topic describes the messages associated with each of the three methods that AWS IoT provides for working with shadows. These methods include the following:

`UPDATE`

Creates a shadow if it doesn't exist, or updates the contents of an existing shadow with the state information provided in the message body. AWS IoT records a timestamp with each update to indicate when the state was last updated. When the shadow's state changes, AWS IoT sends `/delta` messages to all MQTT subscribers with the difference between the `desired` and the `reported` states. Devices or apps that receive a `/delta` message can perform actions based on the difference. For example, a device can update its state to the desired state, or an app can update its UI to reflect the device's state change.

`GET`

Retrieves a current shadow document that contains the complete state of the shadow, including metadata.

`DELETE`

Deletes the device shadow and its content.

You can't restore a deleted device shadow document, but you can create a new device shadow with the name of a deleted device shadow document. If you create a device shadow document that has the same name as one that was deleted within the past 48 hours, the version number of the new device shadow document will follow that of the deleted one. If a device shadow document has been deleted for more than 48 hours, the version number of a new device shadow document with the same name will be 0.

Protocol support

AWS IoT supports MQTT and a REST API over HTTPS protocols to interact with shadows. AWS IoT provides a set of reserved request and response topics for MQTT publish and subscribe actions. Devices and apps should subscribe to the response topics before publishing to a request topic for information about how AWS IoT handled the request. For more information, see Device Shadow MQTT topics and Device Shadow REST API.

Requesting and reporting state

When designing your IoT solution using AWS IoT and shadows, you should determine the apps or devices that will request changes and those that will implement them. Typically, a device implements and reports changes back to the shadow and apps and services respond to and request changes in the shadow. Your solution could be different, but the examples in this topic assume that the client app or service requests changes in the shadow and the device performs the changes and reports them back to the shadow.

Updating a shadow

Your app or service can update a shadow's state by using the UpdateThingShadow API or by publishing to the /update topic. Updates affect only the fields specified in the request.

Updating a shadow when a client requests a state change

When a client requests a state change in a shadow by using the MQTT protocol

1. The client should have a current shadow document so that it can identify the properties to change. See the /get action for how to obtain the current shadow document.
2. The client subscribes to these MQTT topics:
   - `$aws/things/thingName/shadow/name/shadowName/update/accepted`
   - `$aws/things/thingName/shadow/name/shadowName/update/rejected`
   - `$aws/things/thingName/shadow/name/shadowName/update/delta`
   - `$aws/things/thingName/shadow/name/shadowName/update/documents`
3. The client publishes a `$aws/things/thingName/shadow/name/shadowName/update` request topic with a state document that contains the desired state of the shadow. Only the properties to change need to be included in the document. This is an example of a document with the desired state.

```
{
  "state": {
    "desired": {
      "color": {
        "r": 10
      },
      "engine": "ON"
    }
  }
}
```

4. If the update request is valid, AWS IoT updates the desired state in the shadow and publishes messages on these topics:

   - `$aws/things/thingName/shadow/name/shadowName/update/accepted`
   - `$aws/things/thingName/shadow/name/shadowName/update/delta`

   The `/update/accepted` message contains an /accepted response state document shadow document, and the `/update/delta` message contains a /delta response state document shadow document.

5. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/update/rejected` topic with an Error response document shadow document that describes the error.

When a client requests a state change in a shadow by using the API

1. The client calls the `UpdateThingShadow` API with a Request state document state document as its message body.

2. If the request was valid, AWS IoT returns an HTTP success response code and an /accepted response state document shadow document as its response message body.

   AWS IoT will also publish an MQTT message to the `$aws/things/thingName/shadow/name/shadowName/update/delta` topic with a /delta response state document shadow document for any devices or clients that subscribe to it.

3. If the request was not valid, AWS IoT returns an HTTP error response code an Error response document as its response message body.

When the device receives the `/desired` state on the `/update/delta` topic, it makes the desired changes in the device. It then sends a message to the `/update` topic to report its current state to the shadow.

Updating a shadow when a device reports its current state

When a device reports its current state to the shadow by using the MQTT protocol

1. The device should subscribe to these MQTT topics before updating the shadow:
   - `$aws/things/thingName/shadow/name/shadowName/update/accepted`
   - `$aws/things/thingName/shadow/name/shadowName/update/rejected`
   - `$aws/things/thingName/shadow/name/shadowName/update/delta`
   - `$aws/things/thingName/shadow/name/shadowName/update/documents`
2. The device reports its current state by publishing a message to the `$aws/things/thingName/shadow/name/shadowName/update` topic that reports the current state, such as in this example.

```
{
    "state": {
        "reported" : {
            "color" : { "r" : 10 },
            "engine" : "ON"
        }
    }
}
```

3. If AWS IoT accepts the update, it publishes a message to the `$aws/things/thingName/shadow/name/shadowName/update/accepted` topics with an /accepted response state document shadow document.
4. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/update/rejected` topic with an Error response document shadow document that describes the error.

When a device reports its current state to the shadow by using the API

1. The device calls the `UpdateThingShadow` API with a Request state document state document as its message body.

2. If the request was valid, AWS IoT updates the shadow and returns an HTTP success response code with an /accepted response state document shadow document as its response message body.

   AWS IoT will also publish an MQTT message to the `$aws/things/thingName/shadow/name/shadowName/update/delta` topic with a /delta response state document shadow document for any devices or clients that subscribe to it.
3. If the request was not valid, AWS IoT returns an HTTP error response code an Error response document as its response message body.

Optimistic locking

You can use the state document version to ensure you are updating the most recent version of a device's shadow document. When you supply a version with an update request, the service rejects the request with an HTTP 409 conflict response code if the current version of the state document does not match the version supplied. The conflict response code can also occur on any API that modifies `ThingShadow`, including `DeleteThingShadow`.

For example:

Initial document:

```
{
  "state": {
    "desired": {
      "colors": [
        "RED",
        "GREEN",
        "BLUE"
      ]
    }
  },
  "version": 10
}
```

Update: (version doesn't match; this request will be rejected)

```
{
  "state": {
    "desired": {
```

```
      "colors": [
        "BLUE"
      ]
    }
  },
  "version": 9
}
```

Result:

```
{
  "code": 409,
  "message": "Version conflict",
  "clientToken": "426bfd96-e720-46d3-95cd-014e3ef12bb6"
}
```

Update: (version matches; this request will be accepted)

```
{
  "state": {
    "desired": {
      "colors": [
        "BLUE"
      ]
    }
  },
  "version": 10
}
```

Final state:

```
{
  "state": {
    "desired": {
      "colors": [
        "BLUE"
      ]
    }
  },
  "version": 11
}
```

Retrieving a shadow document

You can retrieve a shadow document by using the GetThingShadow API or by subscribing and publishing to the /get topic. This retrieves a complete shadow document, including any delta between the `desired` and `reported` states. The procedure for this task is the same whether the device or a client is making the request.

To retrieve a shadow document by using the MQTT protocol

1. The device or client should subscribe to these MQTT topics before updating the shadow:
   - `$aws/things/thingName/shadow/name/shadowName/get/accepted`
   - `$aws/things/thingName/shadow/name/shadowName/get/rejected`
2. The device or client publishes a message to the `$aws/things/thingName/shadow/name/shadowName/get` topic with an empty message body.
3. If the request is successful, AWS IoT publishes a message to the `$aws/things/thingName/shadow/name/shadowName/get/accepted` topic with a /accepted response state document in the message body.
4. If the request was not valid, AWS IoT publishes a message to the `$aws/things/thingName/shadow/name/shadowName/get/rejected` topic with an Error response document in the message body.

To retrieve a shadow document by using a REST API

1. The device or client call the `GetThingShadow` API with an empty message body.
2. If the request is valid, AWS IoT returns an HTTP success response code with an /accepted response state document shadow document as its response message body.
3. If the request is not valid, AWS IoT returns an HTTP error response code an Error response document as its response message body.

Deleting shadow data

There are two ways to delete shadow data: you can delete specific properties in the shadow document and you can delete the shadow completely.

- To delete specific properties from a shadow, update the shadow; however set the value of the properties that you want to delete to `null`. Fields with a value of `null` are removed from the shadow document.
- To delete the entire shadow, use the DeleteThingShadow API or publish to the /delete topic.

> **Note**
>
> Deleting a shadow doesn't reset its version number to zero at once. It will be reset to zero after 48 hours.

Deleting a property from a shadow document

To delete a property from a shadow by using the MQTT protocol

1. The device or client should have a current shadow document so that it can identify the properties to change. See Retrieving a shadow document for information on how to obtain the current shadow document.
2. The device or client subscribes to these MQTT topics:
   - `$aws/things/`*`thingName`*`/shadow/name/`*`shadowName`*`/update/accepted`
   - `$aws/things/`*`thingName`*`/shadow/name/`*`shadowName`*`/update/rejected`
3. The device or client publishes a `$aws/things/`*`thingName`*`/shadow/name/`*`shadowName`*`/update` request topic with a state document that assigns `null` values to the properties of the shadow to delete. Only the properties to change need to be included in the document. This is an example of a document that deletes the `engine` property.

   ```
   {
     "state": {
       "desired": {
         "engine": null
       }
     }
   }
   ```
4. If the update request is valid, AWS IoT deletes the specified properties in the shadow and publishes a messages with the `$aws/things/`*`thingName`*`/shadow/name/`*`shadowName`*`/`

update/accepted topic with an /accepted response state document shadow document in the message body.

5. If the update request is not valid, AWS IoT publishes a message with the $aws/things/*thingName*/shadow/name/*shadowName*/update/rejected topic with an Error response document shadow document that describes the error.

To delete a property from a shadow by using the REST API

1. The device or client calls the UpdateThingShadow API with a Request state document that assigns null values to the properties of the shadow to delete. Include only the properties that you want to delete in the document. This is an example of a document that deletes the engine property.

```
{
  "state": {
    "desired": {
      "engine": null
    }
  }
}
```

2. If the request was valid, AWS IoT returns an HTTP success response code and an /accepted response state document shadow document as its response message body.
3. If the request was not valid, AWS IoT returns an HTTP error response code an Error response document as its response message body.

Deleting a shadow

Following are some considerations when deleting a device's shadow.

- Setting the device's shadow state to null does not delete the shadow. The shadow version will be incremented on the next update.
- Deleting a device's shadow does not delete the thing object. Deleting a thing object does not delete the corresponding device's shadow.
- Deleting a shadow doesn't reset its version number to zero at once. It will be reset to zero after 48 hours.

To delete a shadow by using the MQTT protocol

1. The device or client subscribes to these MQTT topics:
   - `$aws/things/thingName/shadow/name/shadowName/delete/accepted`
   - `$aws/things/thingName/shadow/name/shadowName/delete/rejected`
2. The device or client publishes a `$aws/things/thingName/shadow/name/shadowName/delete` with an empty message buffer.
3. If the delete request is valid, AWS IoT deletes the shadow and publishes a messages with the `$aws/things/thingName/shadow/name/shadowName/delete/accepted` topic and an abbreviated /accepted response state document shadow document in the message body. This is an example of the accepted delete message:

```
{
  "version": 4,
  "timestamp": 1591057529
}
```

4. If the update request is not valid, AWS IoT publishes a message with the `$aws/things/thingName/shadow/name/shadowName/delete/rejected` topic with an Error response document shadow document that describes the error.

To delete a shadow by using the REST API

1. The device or client calls the `DeleteThingShadow` API with an empty message buffer.
2. If the request was valid, AWS IoT returns an HTTP success response code and an /accepted response state document and an abbreviated /accepted response state document shadow document in the message body. This is an example of the accepted delete message:

```
{
  "version": 4,
  "timestamp": 1591057529
}
```

3. If the request was not valid, AWS IoT returns an HTTP error response code an Error response document as its response message body.

Device Shadow REST API

A shadow exposes the following URI for updating state information:

```
https://account-specific-prefix-ats.iot.region.amazonaws.com/things/thingName/shadow
```

The endpoint is specific to your AWS account. To find your endpoint, you can:

- Use the describe-endpoint command from the AWS CLI.
- Use the AWS IoT console settings. In **Settings**, the endpoint is listed under **Custom endpoint**
- Use the AWS IoT console thing details page. In the console:
  1. Open **Manage** and under **Manage**, choose **Things**.
  2. In the list of things, choose the thing for which you want to get the endpoint URI.
  3. Choose the **Device Shadows** tab and choose your shadow. You can view the endpoint URI in the **Device Shadow URL** section of the **Device Shadow details** page.

The format of the endpoint is as follows:

```
identifier.iot.region.amazonaws.com
```

The shadow REST API follows the same HTTPS protocols/port mappings as described in Device communication protocols.

> **Note**
>
> To use the APIs, you must use `iotdevicegateway` as the service name for authentication. For more information, see IoTDataPlane.

API actions

- GetThingShadow
- UpdateThingShadow
- DeleteThingShadow
- ListNamedShadowsForThing

You can also use the API to create a named shadow by providing `name=shadowName` as part of the query parameter of the API.

GetThingShadow

Gets the shadow for the specified thing.

The response state document includes the delta between the `desired` and `reported` states.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP GET https://endpoint/things/thingName/shadow?name=shadowName
Request body: (none)
```

The `name` query parameter is not required for unnamed (classic) shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
Response Body: response state document
```

For more information, see Example Response State Document.

Authorization

Retrieving a shadow requires a policy that allows the caller to perform the `iot:GetThingShadow` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to retrieve a device's shadow:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
                "Effect": "Allow",
                "Action": "iot:GetThingShadow",
                "Resource": [
                    "arn:aws:iot:us-east-1:123456789012:thing/thing"
                ]
            }
        ]
}
```

UpdateThingShadow

Updates the shadow for the specified thing.

Updates affect only the fields specified in the request state document. Any field with a value of `null` is removed from the device's shadow.

Request

The request includes the standard HTTP headers plus the following URI and body:

```
HTTP POST https://endpoint/things/thingName/shadow?name=shadowName
Request body: request state document
```

The `name` query parameter is not required for unnamed (classic) shadows.

For more information, see Example Request State Document.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
Response body: response state document
```

For more information, see Example Response State Document.

Authorization

Updating a shadow requires a policy that allows the caller to perform the `iot:UpdateThingShadow` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to update a device's shadow:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:UpdateThingShadow",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:thing/thing"
            ]
        }
    ]
}
```

DeleteThingShadow

Deletes the shadow for the specified thing.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP DELETE https://endpoint/things/thingName/shadow?name=shadowName
Request body: (none)
```

The `name` query parameter is not required for unnamed (classic) shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
Response body: Empty response state document
```

Note that deleting a shadow does not reset its version number to 0.

Authorization

Deleting a device's shadow requires a policy that allows the caller to perform the `iot:DeleteThingShadow` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to delete a device's shadow:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:DeleteThingShadow",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:thing/thing"
            ]
        }
    ]
}
```

ListNamedShadowsForThing

Lists the shadows for the specified thing.

Request

The request includes the standard HTTP headers plus the following URI:

```
HTTP GET /api/things/shadow/ListNamedShadowsForThing/thingName?
nextToken=nextToken&pageSize=pageSize
Request body: (none)
```

nextToken

The token to retrieve the next set of results.

This value is returned on paged results and is used in the call that returns the next page.

pageSize

The number of shadow names to return in each call. See also `nextToken`.

thingName

The name of the thing for which to list the named shadows.

Response

Upon success, the response includes the standard HTTP headers plus the following response code and a Shadow name list response document.

> **Note**
>
> The unnamed (classic) shadow does not appear in this list. The response is an empty list if you only have a classic shadow or if the `thingName` you specify doesn't exist.

```
HTTP 200
Response body: Shadow name list document
```

Authorization

Listing a device's shadow requires a policy that allows the caller to perform the `iot:ListNamedShadowsForThing` action. The Device Shadow service accepts two forms of authentication: Signature Version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to list a thing's named shadows:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:ListNamedShadowsForThing",
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:thing/thing"
            ]
        }
    ]
}
```

Device Shadow MQTT topics

The Device Shadow service uses reserved MQTT topics to enable devices and apps to get, update, or delete the state information for a device (shadow).

Publishing and subscribing on shadow topics requires topic-based authorization. AWS IoT reserves the right to add new topics to the existing topic structure. For this reason, we recommend that you avoid wild card subscriptions to shadow topics. For example, avoid subscribing to topic filters like `$aws/things/thingName/shadow/#` because the number of topics that match this topic filter might increase as AWS IoT introduces new shadow topics. For examples of the messages published on these topics see Interacting with shadows.

Shadows can be named or unnamed (classic). The topics used by each differ only in the topic prefix. This table shows the topic prefix used by each shadow type.

ShadowTopicPrefix **value**	**Shadow type**
`$aws/things/` *`thingName`* `/shadow`	Unnamed (classic) shadow
`$aws/things/` *`thingName`* `/shadow/name/` *`shadowName`*	Named shadow

To create a complete topic, select the *`ShadowTopicPrefix`* for the type of shadow to which you want to refer, replace *`thingName`*, and *`shadowName`* if applicable, with their corresponding values, and then append that with the topic stub as shown in the following sections.

The following are the MQTT topics used for interacting with shadows.

Topics

- /get
- /get/accepted
- /get/rejected
- /update
- /update/delta
- /update/accepted
- /update/documents

- /update/rejected
- /delete
- /delete/accepted
- /delete/rejected

/get

Publish an empty message to this topic to get the device's shadow:

```
ShadowTopicPrefix/get
```

AWS IoT responds by publishing to either /get/accepted or /get/rejected.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/get"
            ]
        }
    ]
}
```

/get/accepted

AWS IoT publishes a response shadow document to this topic when returning the device's shadow:

```
ShadowTopicPrefix/get/accepted
```

For more information, see Response state documents.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/get/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/get/accepted"
            ]
        }
    ]
}
```

/get/rejected

AWS IoT publishes an error response document to this topic when it can't return the device's shadow:

```
ShadowTopicPrefix/get/rejected
```

For more information, see Error response document.

Example policy

The following is an example of the required policy:

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/thingName/
shadow/get/rejected"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/shadow/get/
rejected"
      ]
    }
  ]
}
```

/update

Publish a request state document to this topic to update the device's shadow:

```
ShadowTopicPrefix/update
```

The message body contains a partial request state document.

A client attempting to update the state of a device would send a JSON request state document with the `desired` property such as this:

```
{
  "state": {
    "desired": {
      "color": "red",
      "power": "on"
    }
  }
}
```

A device updating its shadow would send a JSON request state document with the `reported` property, such as this:

```
{
  "state": {
    "reported": {
      "color": "red",
      "power": "on"
    }
  }
}
```

AWS IoT responds by publishing to either [/update/accepted]() or [/update/rejected]().

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/update"
            ]
        }
    ]
```

```
}
```

/update/delta

AWS IoT publishes a response state document to this topic when it accepts a change for the device's shadow, and the response state document contains different values for `desired` and `reported` states:

```
ShadowTopicPrefix/update/delta
```

The message buffer contains a /delta response state document.

Message body details

- A message published on `update/delta` includes only the desired attributes that differ between the `desired` and `reported` sections. It contains all of these attributes, regardless of whether these attributes were contained in the current update message or were already stored in AWS IoT. Attributes that do not differ between the `desired` and `reported` sections are not included.
- If an attribute is in the `reported` section but has no equivalent in the `desired` section, it is not included.
- If an attribute is in the `desired` section but has no equivalent in the `reported` section, it is included.
- If an attribute is deleted from the `reported` section but still exists in the `desired` section, it is included.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
```

```
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/update/delta"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/update/delta"
            ]
        }
    ]
}
```

/update/accepted

AWS IoT publishes a response state document to this topic when it accepts a change for the device's shadow:

```
ShadowTopicPrefix/update/accepted
```

The message buffer contains a /accepted response state document.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
```

```
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/update/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/update/accepted"
            ]
        }
    ]
}
```

/update/documents

AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed:

```
ShadowTopicPrefix/update/documents
```

The message body contains a /documents response state document.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
```

```
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/update/documents"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/update/documents"
            ]
        }
    ]
}
```

/update/rejected

AWS IoT publishes an error response document to this topic when it rejects a change for the device's shadow:

```
ShadowTopicPrefix/update/rejected
```

The message body contains an Error response document.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/update/rejected"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/update/rejected"
            ]
        }
    ]
}
```

/delete

To delete a device's shadow, publish an empty message to the delete topic:

```
ShadowTopicPrefix/delete
```

The content of the message is ignored.

Note that deleting a shadow does not reset its version number to 0.

AWS IoT responds by publishing to either /delete/accepted or /delete/rejected.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
```

```
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/delete"
            ]
        }
    ]
}
```

/delete/accepted

AWS IoT publishes a message to this topic when a device's shadow is deleted:

```
ShadowTopicPrefix/delete/accepted
```

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/delete/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/delete/accepted"
            ]
        }
```

```
    ]
}
```

/delete/rejected

AWS IoT publishes an error response document to this topic when it can't delete the device's shadow:

```
ShadowTopicPrefix/delete/rejected
```

The message body contains an Error response document.

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
things/thingName/shadow/delete/rejected"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thingName/
shadow/delete/rejected"
            ]
        }
    ]
```

```
}
```

Device Shadow service documents

The Device Shadow service respects all rules of the JSON specification. Values, objects, and arrays are stored in the device's shadow document.

Contents

Shadow document examples

The Device Shadow service uses these documents in UPDATE, GET, and DELETE operations using the REST API or MQTT Pub/Sub Messages.

Examples

Request state document

A request state document has the following format:

```
{
    "state": {
        "desired": {
            "attribute1": integer2,
```

```
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        },
        "reported": {
            "attribute1": integer1,
            "attribute2": "string1",
            ...
            "attributeN": boolean1
        }
    },
    "clientToken": "token",
    "version": version
}
```

- `state` — Updates affect only the fields specified. Typically, you'll use either the `desired` or the `reported` property, but not both in the same request.
  - `desired` — The state properties and values requested to be updated in the device.
  - `reported` — The state properties and values reported by the device.
- `clientToken` — If used, you can match the request and corresponding response by the client token.
- `version` — If used, the Device Shadow service processes the update only if the specified version matches the latest version it has.

Response state documents

Response state documents have the following format depending on the response type.

/accepted response state document

```
{
    "state": {
        "desired": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        }
    },
    "metadata": {
```

```
        "desired": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        }
    },
    "timestamp": timestamp,
    "clientToken": "token",
    "version": version
}
```

/delta response state document

```
{
    "state": {
        "attribute1": integer2,
        "attribute2": "string2",
        ...
        "attributeN": boolean2
    },
    "metadata": {
        "attribute1": {
            "timestamp": timestamp
        },
        "attribute2": {
            "timestamp": timestamp
        },
        ...
        "attributeN": {
            "timestamp": timestamp
        }
    },
    "timestamp": timestamp,
    "clientToken": "token",
    "version": version
}
```

/documents response state document

```
{
  "previous" : {
    "state": {
        "desired": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        },
        "reported": {
            "attribute1": integer1,
            "attribute2": "string1",
            ...
            "attributeN": boolean1
        }
    },
    "metadata": {
        "desired": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        },
        "reported": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        }
    },
```

```
    "version": version-1
  },
  "current": {
    "state": {
        "desired": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        },
        "reported": {
            "attribute1": integer2,
            "attribute2": "string2",
            ...
            "attributeN": boolean2
        }
    },
    "metadata": {
        "desired": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        },
        "reported": {
            "attribute1": {
                "timestamp": timestamp
            },
            "attribute2": {
                "timestamp": timestamp
            },
            ...
            "attributeN": {
                "timestamp": timestamp
            }
        }
    },
    "version": version
```

```
  },
  "timestamp": timestamp,
  "clientToken": "token"
}
```

Response state document properties

- `previous` — After a successful update, contains the `state` of the object before the update.
- `current` — After a successful update, contains the `state` of the object after the update.
- `state`
  - `reported` — Present only if a thing reported any data in the `reported` section and contains only fields that were in the request state document.
  - `desired` — Present only if a device reported any data in the `desired` section and contains only fields that were in the request state document.
- `metadata` — Contains the timestamps for each attribute in the `desired` and `reported` sections so that you can determine when the state was updated.
- `timestamp` — The Epoch date and time the response was generated by AWS IoT.
- `clientToken` — Present only if a client token was used when publishing valid JSON to the `/update` topic.
- `version` — The current version of the document for the device's shadow shared in AWS IoT. It is increased by one over the previous version of the document.

Error response document

An error response document has the following format:

```
{
    "code": error-code,
    "message": "error-message",
    "timestamp": timestamp,
    "clientToken": "token"
}
```

- `code` — An HTTP response code that indicates the type of error.
- `message` — A text message that provides additional information.

- `timestamp` — The date and time the response was generated by AWS IoT. This property is not present in all error response documents.
- `clientToken` — Present only if a client token was used in the published message.

For more information, see Device Shadow error messages.

Shadow name list response document

A shadow name list response document has the following format:

```
{
    "results": [
        "shadowName-1",
        "shadowName-2",
        "shadowName-3",
        "shadowName-n"
    ],
    "nextToken": "nextToken",
    "timestamp": timestamp
}
```

- `results` — The array of shadow names.
- `nextToken` — The token value to use in paged requests to get the next page in the sequence. This property is not present when there are no more shadow names to return.
- `timestamp` — The date and time the response was generated by AWS IoT.

Document properties

A device's shadow document has the following properties:

`state`

`desired`

The desired state of the device. Apps can write to this portion of the document to update the state of a device directly without having to connect to it.

`reported`

The reported state of the device. Devices write to this portion of the document to report their new state. Apps read this portion of the document to determine the device's last-reported state.

`metadata`

Information about the data stored in the `state` section of the document. This includes timestamps, in Epoch time, for each attribute in the `state` section, which enables you to determine when they were updated.

> **Note**
>
> Metadata do not contribute to the document size for service limits or pricing. For more information, see AWS IoT Service Limits.

`timestamp`

Indicates when the message was sent by AWS IoT. By using the timestamp in the message and the timestamps for individual attributes in the `desired` or `reported` section, a device can determine a property's age, even if the device doesn't have an internal clock.

`clientToken`

A string unique to the device that enables you to associate responses with requests in an MQTT environment.

`version`

The document version. Every time the document is updated, this version number is incremented. Used to ensure the version of the document being updated is the most recent.

For more information, see Shadow document examples.

Delta state

Delta state is a virtual type of state that contains the difference between the `desired` and `reported` states. Fields in the `desired` section that are not in the `reported` section are included in the delta. Fields that are in the `reported` section and not in the `desired` section are not included in the delta. The delta contains metadata, and its values are equal to the metadata in the `desired` field. For example:

```
{
  "state": {
    "desired": {
      "color": "RED",
      "state": "STOP"
    },
    "reported": {
      "color": "GREEN",
      "engine": "ON"
    },
    "delta": {
      "color": "RED",
      "state": "STOP"
    }
  },
  "metadata": {
    "desired": {
      "color": {
        "timestamp": 12345
      },
      "state": {
        "timestamp": 12345
      }
      },
      "reported": {
        "color": {
          "timestamp": 12345
        },
        "engine": {
          "timestamp": 12345
        }
      },
      "delta": {
        "color": {
          "timestamp": 12345
        },
        "state": {
          "timestamp": 12345
        }
      }
```

```
    },
    "version": 17,
    "timestamp": 123456789
  }
}
```

When nested objects differ, the delta contains the path all the way to the root.

```
{
  "state": {
    "desired": {
      "lights": {
        "color": {
          "r": 255,
          "g": 255,
          "b": 255
        }
      }
    },
    "reported": {
      "lights": {
        "color": {
          "r": 255,
          "g": 0,
          "b": 255
        }
      }
    },
    "delta": {
      "lights": {
        "color": {
          "g": 255
        }
      }
    }
  },
  "version": 18,
  "timestamp": 123456789
}
```

The Device Shadow service calculates the delta by iterating through each field in the `desired` state and comparing it to the `reported` state.

Arrays are treated like values. If an array in the `desired` section doesn't match the array in the `reported` section, then the entire desired array is copied into the delta.

Versioning shadow documents

The Device Shadow service supports versioning on every update message, both request and response. This means that with every update of a shadow, the version of the JSON document is incremented. This ensures two things:

- A client can receive an error if it attempts to overwrite a shadow using an older version number. The client is informed it must resync before it can update a device's shadow.
- A client can decide not to act on a received message if the message has a lower version than the version stored by the client.

A client can bypass version matching by not including a version in the shadow document.

Client tokens in shadow documents

You can use a client token with MQTT-based messaging to verify the same client token is contained in a request and request response. This ensures the response and request are associated.

Note

The client token can be no longer than 64 bytes. A client token that is longer than 64 bytes causes a 400 (Bad Request) response and an *Invalid clientToken* error message.

Empty shadow document properties

The `reported` and `desired` properties in a shadow document can be empty or omitted when they don't apply to the current shadow state. For example, a shadow document contains a `desired` property only if it has a desired state. The following is a valid example of a state document with no `desired` property:

```
{
    "reported" : { "temp": 55 }
}
```

The `reported` property can also be empty, such as if the shadow has not been updated by the device:

```
{
    "desired" : { "color" : "RED" }
}
```

If an update causes the `desired` or `reported` properties to become null, it is removed from the document. The following shows how to remove the `desired` property by setting it to `null`. You might do this when a device updates its state, for example.

```
{
    "state": {
        "reported": {
            "color": "red"
        },
        "desired": null
    }
}
```

A shadow document can also have neither `desired` or `reported` properties, making the shadow document empty. This is an example of an empty, yet valid shadow document.

```
{
}
```

Array values in shadow documents

Shadows support arrays, but treat them as normal values in that an update to an array replaces the whole array. It is not possible to update part of an array.

Initial state:

```
{
    "desired" : { "colors" : ["RED", "GREEN", "BLUE" ] }
}
```

Update:

```
{
```

```
    "desired" : { "colors" : ["RED"] }
}
```

Final state:

```
{
    "desired" : { "colors" : ["RED"] }
}
```

Arrays can't have null values. For example, the following array is not valid and will be rejected.

```
{
    "desired" : {
        "colors" : [ null, "RED", "GREEN" ]
    }
}
```

Device Shadow error messages

The Device Shadow service publishes a message on the error topic (over MQTT) when an attempt to change the state document fails. This message is only emitted as a response to a publish request on one of the reserved `$aws` topics. If the client updates the document using the REST API, then it receives the HTTP error code as part of its response, and no MQTT error messages are emitted.

HTTP error code	Error messages
400 (Bad Request)	• Invalid JSON • Missing required node: state • State node must be an object • Desired node must be an object • Reported node must be an object • Invalid version • Invalid clientToken

HTTP error code	Error messages
	Note A client token that is longer than 64 bytes will cause this response. • JSON contains too many levels of nesting; maximum is 6 • State contains an invalid node
401 (Unauthorized)	• Unauthorized
403 (Forbidden)	• Forbidden
404 (Not Found)	• Thing not found • No shadow exists with name: *shadowName*
409 (Conflict)	• Version conflict
413 (Payload Too Large)	• The payload exceeds the maximum size allowed
415 (Unsupported Media Type)	• Unsupported documented encoding; supported encoding is UTF-8
429 (Too Many Requests)	• The Device Shadow service will generate this error message when there are more than 10 in-flight requests on a single connection. An in-flight request is an in-progress request that has been started but not yet completed.
500 (Internal Server Error)	• Internal service failure

AWS IoT Device Management Software Package Catalog

With AWS IoT Device Management Software Package Catalog, you can maintain an inventory of software packages and their versions. You can associate package versions to individual things and AWS IoT dynamic thing groups, and deploy them through in-house processes or AWS IoT jobs.

A software package contains one or more package versions, which is a collection of files that can be deployed as a single unit. Package versions can contain firmware, operating system updates, device applications, configurations, and security patches. As the software evolves over time, you can create a new package version and deploy it to your fleet.

The AWS IoT software package hub is located within AWS IoT Core. You can use the hub to centrally register and maintain your software package inventory and metadata, which creates a catalog of software packages and their versions. You can choose to group devices based on software packages and package versions deployed on the device. This feature provides the opportunity to keep device-side package inventory as a named shadow, associate and group devices based on versions, and visualize package version distribution across the fleet by using fleet metrics.

If you have an in-house software deployment system established, you can continue to use that process to deploy your package versions. If you don't have a deployment process established or if you prefer, we recommend using AWS IoT jobs to use the features in the Software Package Catalog. For more information, see Preparing AWS IoT jobs.

This chapter contains the following sections:

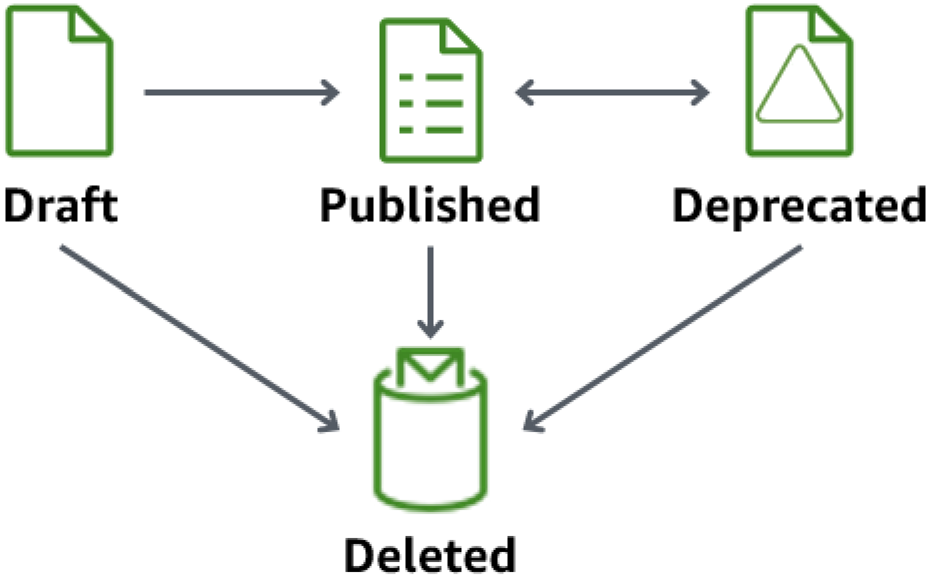
- Preparing to use Software Package Catalog
- Preparing security
- Preparing fleet indexing
- Preparing AWS IoT Jobs
- Getting started with Software Package Catalog

Preparing to use Software Package Catalog

The following section provides an overview of the package version lifecycle and information for using AWS IoT Device Management Software Package Catalog.

Package version lifecycle

A package version can evolve through the following lifecycle states: `draft`, `published`, and `deprecated`. It can also be `deleted`.



- **Draft**

  When you create a package version, it's in a `draft` state. This state indicates that the software package is being prepared or is incomplete.

  While the package version in this state, you can't deploy it. You can edit the package version's description, attributes, and tags.

  You can transition a package version that's in the `draft` state to `published` or be `deleted` by using the console, or by issuing either the UpdatePackageVersion or DeletePackageVersion API operations.

- **Published**

  When your package version is ready to deploy, transition the package version to a `published` state. While in this state, you can choose to identify the package version as the default version by editing the software package in the console or through the UpdatePackage API operation. In this state, you can edit only the description and tags.

You can transition a package version that's in the `published` state to `deprecated` or be `deleted` by using the console, or issuing either the [UpdatePackageVersion] or [DeletePackageVersion] API operations.

- **Deprecated**

  If a new package version is available, you can transition earlier package versions to `deprecated`. You can still deploy jobs with a deprecated package version. You can also name a deprecated package version as the default version, and edit only the description and tags.

  Consider transitioning a package version to `deprecated` when the version is outdated, but you still have devices in the field using the older version or needs to maintain it due to run-time dependency.

  You can transition a package version that's in the `deprecated` state to `published` or be `deleted` by using the console, or issuing either the [UpdatePackageVersion] or [DeletePackageVersion] API operattions.

- **Deleted**

  When you no longer intend to use a package version, you can delete it by using the console or issuing the [DeletePackageVersion] API operation.

> **Note**
>
> If you delete a package version while there are pending jobs that reference it, you will receive an error message when the job successfully completes and attempts to update the reserved named shadow.
> If the software package version you want to delete is named as the default package version, you must first update the package to name another version as default or leave the field unnamed. You can do this by using the console or the [UpdatePackageVersion] API operation. (To remove any named package version as default, set the [unsetDefaultVersion] parameter to true when you issue the [UpdatePackage] API operation).

If you delete a software package through the console, it deletes all of the package versions associated with that package, unless one is named as the default version.

Package version naming conventions

When you name package versions, it's important to plan and apply a logical naming strategy so that you and others can easily identify the latest package version and the version progression. You must provide a version name when creating the package version, but the strategy and format is largely up to your business case.

As a best practice, we recommend using the Semantic Versioning SemVer format. For example, `1.2.3` where `1` is the major version for functionally incompatible changes, `2` the major version for functionally compatible changes, and `3` is the patch version (for bug fixes). For more information, see Semantic Versioning 2.0.0. For more information about the package version name requirements, see versionName in the AWS IoT API reference guide.

Default version

Setting a version as default is optional. You can add or remove default package versions. You can also deploy a package version that is not named as the default version.

When you create a package version, it’s placed in a `draft` state and can’t be named as the default version until you transition the package version to published. Software Package Catalog doesn’t automatically select a version as default or update a newer package version as the default. You must intentionally name the package version you choose through the console or by issuing the UpdatePackageVersion API operation.

Version attributes

Version attributes and their values hold important information about your package versions. We recommend that you define general purpose attributes for a package or package version. For example, you might create a name-value pair for platform, architecture, operating system, release date, author, or Amazon S3 URL.

When you create an AWS IoT job with a job document, you can also choose to use a substitution variable (`$parameter`) that refers to an attribute’s value. For more information, see Preparing AWS IoT Jobs.

Version attributes that are used in package versions will not be automatically added to the reserved named shadow and can’t be indexed or queried through Fleet Indexing directly. To index or query package version attributes through Fleet Indexing, you can populate the version attribute in the reserved named shadow.

We recommend that the version attribute parameter in the reserved named shadow capture device-reported properties , such as operation system and installation time. They can also be indexed and queried through Fleet Indexing.

Version attributes aren't required to follow a specific naming convention. You can create name-value pairs to meet your business needs. The combined size of all the attributes on a package version is limited to 3KB. For more information, see Software Package Catalog software package and package versions limits.

Using all attributes in a job document

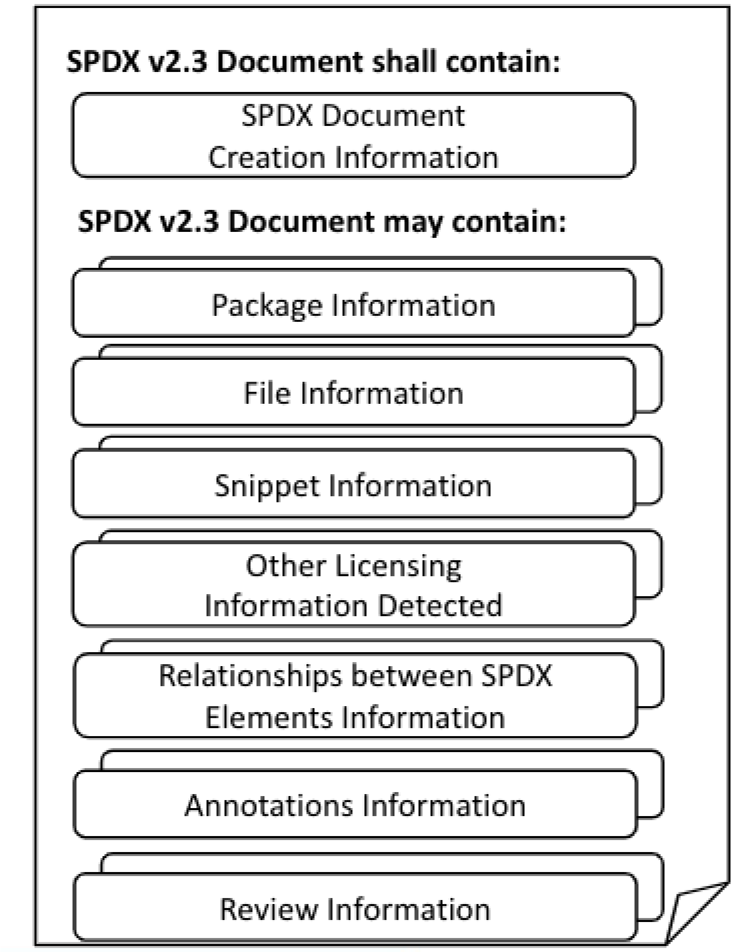
You can have all package version attributes automatically added to your job deployment for selected devices. To automatically use all package version attributes programmatically in an API or CLI command, refer to the following job document example:

```
"TestPackage": "${aws:iot:package:TestPackage:version:PackageVersion:attributes}"
```

Software Bill of Materials

The software bill of materials (SBOM) provides a central repository for all aspects of your software package. In addition to storing software packages and package versions, you can store the software bill of materials (SBOM) associated with each package version in AWS IoT Device Management Software Package Catalog. A software package contains one or more package versions and each package version consists of one or more components. Each of those components supporting the composition of a specific package version can be described and cataloged using a software bill of materials. The industry standards for software bill of materials supported are SPDX and CycloneDX. When a SBOM is first created, it undergoes validation against the SPDX and CycloneDX industry standard format. For more information on SPDX, see System Package Data Exchange. For more information on CycloneDX, see CycloneDX.

The software bill of materials describes all aspects of a specific package version's components such as package information, file information, and other pertinent metadata. See the below example of a software bill of materials document structure in the SPDX format:



Software Bill of Materials Benefits

One of the key benefits for adding your software bill of materials for a package version in Software Package Catalog is vulnerability management.

Vulnerability management

Assessing and mitigating your vulnerability to apparent security risks in software components remains critical to protecting the integrity of your device fleet. With the addition of the software bill of materials stored in Software Package Catalog for each package version, you can proactively expose gaps in security by knowing what devices are at risk based on their package version and SBOM using your own in-house vulnerability management solution. You can deploy fixes to those affected devices and protect your fleet of devices.

Software Bill of Materials Storage

The software bill of materials (SBOM) for each software package version are stored in an Amazon S3 bucket using the Amazon S3 versioning feature. The Amazon S3 bucket storing the SBOM must be located in the same region where the package version was created. An Amazon S3 bucket using the versioning feature maintains multiple variants of an object in the same bucket. For more information on using versioning in an Amazon S3 bucket, see Using versioning in Amazon S3 buckets.

> **Note**
>
> Each software package version can have multiple SBOM files attached to it, but the SBOM files must be stored in a single zip archive file.

The specific Amazon S3 key and version ID for your bucket are used to uniquely identify each version of a software bill of materials for a package version.

> **Note**
>
> For a package version with a single SBOM file, you can store that SBOM file in your Amazon S3 bucket as a zip archive file.
> For a package version with multiple SBOM files, you must place all SBOM files in a single zip archive file and then store that zip archive file in your Amazon S3 bucket.
> All SBOM files stored in the single zip archive file in both scenarios are formatted as either SPDX or CycloneDX .json files.

Permissions Policy

In order for AWS IoT acting as the specified principal to access the SBOM zip archive files stored in the Amazon S3 bucket, you need a resource-based permissions policy. Refer to the following example for the correct resource-based permissions policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "iot.amazonaws.com"
                ]
        },
            "Action": "s3:*",
            "Resource": "arn:aws:s3:::bucketName/*"
        }
    ]
}
```

For more information on resource-based permission policies, see AWS IoT resource-based policies

Updating the SBOM

You can update the software bill of materials as often as needed to protect and enhance your device fleet. Each time the software bill of materials is updated in your Amazon S3 bucket, the version ID changes and you must associate the new Amazon S3 bucket URL with the appropriate software package version. You will see the new version ID in the **Amazon S3 Object version ID** column on the package version page in the AWS Management Console. Additionally, you can use the API operation `GetPackageVersion` or the CLI command `get-package-version` to view the new version ID.

> **Note**
>
> Updating your software bill of materials, which will cause a new version ID, will not cause a new package version to be created.

For more information on Amazon S3 object keys, see Creating object key names.

Enabling AWS IoT fleet indexing

Enabling AWS IoT fleet indexing is a requirement to use AWS IoT Device Management Software Package Catalog. To leverage AWS IoT fleet indexing with Software Package Catalog, set the reserved named shadow (`$package`) as the data source for each device you want to index on and gather metrics. For more information on reserved named shadows, see Reserved named shadow.

Fleet indexing provides support that enables AWS IoT things to be grouped through dynamic thing groups that are filtered by software package version. For example, fleet indexing can identify things that have or don't have a specific package version installed, don't have any package versions installed, or match specific name-value pairs. Finally, fleet indexing provides standard and custom metrics that you can use to gain insight about the state of your device fleet. For more information, see Preparing fleet indexing.

> **Note**
>
> Enabling fleet indexing for Software Package Catalog incurs standard service costs. For more information, see AWS IoT Device Management, Pricing.

Reserved named shadow

The reserved named shadow, `$package`, reflects the state of the device's installed software packages and package versions. Fleet indexing uses the reserved named shadow as a data source to build standard and custom metrics so you can query the state of your fleet. For more information, see Preparing fleet indexing.

A reserved named shadow is similar to a named shadow with the exception that its name is predefined and you can't change it. In addition, the reserved named shadow doesn't update with metadata, and uses only the `version` and `attributes` keywords.

Update requests that include other keywords, such as `description`, will receive an error response under the `rejected` topic. For more information, see Device Shadow error messages.

It can be created when you create an AWS IoT thing through the console, when an AWS IoT job successfully completes and updates the shadow, and if you issue the `UpdateThingShadow` API operation. For more information, see UpdateThingShadow in the AWS IoT Core developer guide.

> **Note**
>
> Indexing the reserved named shadow doesn't count toward the number of named shadows that fleet indexing can index. For more information, see AWS IoT Device Management fleet indexing limits and quotas. In addition, if you choose to have AWS IoT jobs update the reserved named shadow when a job successfully completes, the API call is counted toward

your Device Shadow and registry operations and can incur a cost. For more information, see AWS IoT Device Management jobs limits and quotas and the IndexingFilter API data type.

Structure of the $package shadow

The reserved named shadow contains the following:

```
{
    "state": {
        "reported": {
            "<packageName>": {
                "version": "",
                "attributes": {
                }
            }
        }
    },
    "version" : 1
    "timestamp" : 1672531201
}
```

The shadow properties are updated with the following information:

- `<packageName>`: The name of the installed software package, which is updated with the packageName parameter.
- `version`: The name of the installed package version, which is updated with the versionName parameter.
- `attributes`: Optional metadata stored by the device and indexed by Fleet indexing. This allows customers to query their indexes based on the data stored.
- `version`: The shadow's version number. It's automatically incremented each time the shadow is updated and begins at 1.
- `timestamp`: Indicates when the shadow was last updated and is recorded in Unix time.

For more information about the format and behavior of a named shadow, see AWS IoT Device Shadow service Message order.

Deleting a software package and its package versions

Before you delete a software package, do the following:

- Confirm that the package and its versions aren't actively being deployed.
- Delete all the associated versions first. If one of the versions is designated as the **default version**, you must remove the named default version from the package. Because designating a default version is optional, there is no conflict removing it. To remove the default version from the software package, edit the package through the console or use the UpdatePackageVersion API operation.

As long as there is no named default package version, you can use the console to delete a software package and all of its package versions will also be deleted. If you use an API call to delete software packages, you must delete the package versions first and then the software package.

Preparing security

This section discusses the main security requirements for AWS IoT Device Management Software Package Catalog.

Resource-based authentication

Software Package Catalog uses resource-based authorization to provide added security when updating software on your fleet. This means that you must create an AWS Identity and Access Management (IAM) policy that grants rights to perform `create`, `read`, `update`, `delete`, and `list` actions for software packages and package versions, and reference the specific software packages and package versions that you want to deploy in the `Resources` section. You also need these rights so that you can update the reserved named shadow. You reference the software packages and package versions by including an Amazon Resource Name (ARN) for each entity.

> **Note**
>
> If you intend the policy to grant rights for package version API calls (such as CreatePackageVersion, UpdatePackageVersion, DeletePackageVersion), then you need to include *both* the software package and the package version ARNs in the policy. If you intend the policy to grant rights for software package API calls (such as CreatePackage,

UpdatePackage, and DeletePackage) then you must include only the software package ARN in the policy.

Structure the software package and package version ARNs as follows:

- Software package: `arn:aws:iot:<region>:<accountID>:package/<packageName>/package`
- Package version: `arn:aws:iot:<region>:<accountID>:package/<packageName>/version/<versionName>`

Note

There are other related rights that you might include in this policy. For example, you might include an ARN for the `job`, `thinggroup`, and `jobtemplate`. For more information and a complete listing of the policy options, see Securing users and devices with AWS IoT Jobs.

For example, if you have a software package and package version that's named as follows:

- AWS IoT thing: `myThing`
- Package name: `samplePackage`
- Version `1.0.0`

The policy might look like the following example:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:createPackage",
                "iot:createPackageVersion",
                "iot:updatePackage",
                "iot:updatePackageVersion"
            ],
```

```
            "Resource": [
               "arn:aws:iot:us-east-1:111122223333:package/samplePackage",
               "arn:aws:iot:us-east-1:111122223333:package/samplePackage/
version/1.0.0"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow"
            ],
            "Resource": "arn:aws:iot:us-east-1:111122223333:thing/myThing/$package"
        }
    ]
}
```

AWS IoT Job rights to deploy package versions

For security purposes it's important for you to grant rights to deploy packages and package versions, and name the specific packages and package versions they're allowed to deploy. To do this, you create an IAM role and policy that grants permission to deploy jobs with package versions. The policy must specify the destination package versions as a resource.

IAM policy

The IAM policy grants the right to create a job that includes the package and version that are named in the `Resource` section.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:CreateJob",
                "iot:CreateJobTemplate"
            ],
            "Resource":[
                "arn:aws:iot:*:111122223333:job/<jobId>",
                "arn:aws:iot:*:111122223333:thing/<thingName>/$package",
```

```
                    "arn:aws:iot:*:111122223333:thinggroup/<thingGroupName>",
                    "arn:aws:iot:*:111122223333:jobtemplate/<jobTemplateName>",
                    "arn:aws:iot:*:111122223333:package/<packageName>/
version/<versionName>"
               ]
          }
     ]
}
```

> **Note**
>
> If you want to deploy a job that uninstalls a software package and package version, you must authorize an ARN where the package version is `$null`, such as in the following:

```
arn:aws:iot:<regionCode>:111122223333:package/<packageName>/version/$null
```

AWS IoT Job rights to update the reserved named shadow

To allow jobs to update the thing's reserved name shadow when the job successfully completes, you must create an IAM role and policy. There are two ways you can do this in the AWS IoT console. The first is when you create a software package in the console. If you see an **Enable dependencies for package management** dialog box, you can choose to use an existing role or create a new role. Or, in the AWS IoT console, choose **Settings**, choose **Manage indexing**, and then **Manage indexing for device packages and versions**.

> **Note**
>
> If you choose to have the AWS IoT Job service update the reserved named shadow when a job successfully completes, the API call is counted toward your **Device Shadow and registry operations** and can incur a cost. For more information, see [AWS IoT Core pricing](#).

When you use the **Create role** option, the generated role's name begins with `aws-iot-role-update-shadows` and contains the following policies:

Setting up a role

Permissions

The permissions policy grants the rights to query and update the thing shadow. The `$package` parameter in the resource ARN targets the reserved named shadow.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:DescribeEndpoint",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:111122223333:thing/<thingName>/$package"
            ]
        }
    ]
}
```

Trust relationship

In addition to the permissions policy, the role requires a trust relationship with AWS IoT Core so that the entity can assume the role and update the reserved named shadow.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "iot.amazonaws.com"
        },
            "Action": "sts:AssumeRole"
        }
    ]
```

```
}
```

Setting up a user policy

iam:PassRole permission

Finally, you must have the permission to pass the role to AWS IoT Core when you call the UpdatePackageConfiguration API operation.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole",
                "iot:UpdatePackageConfiguration"
            ],
            "Resource": "arn:aws:iam::111122223333:role/<roleName>"
        }
    ]
}
```

AWS IoT Jobs permissions to download from Amazon S3

The job document is saved in Amazon S3. You refer to this file when you dispatch through AWS IoT Jobs. You must provide AWS IoT Jobs with the rights to download the file (`s3:GetObject`). You must also set up a trust relationship between Amazon S3 and AWS IoT Jobs. For instructions to create these policies, see Presigned URLs in Managing Jobs.

Permissions to update the software bill of materials for a package version

To update the software bill of materials for a package version in the `Draft`, `Published`, or `Deprecated` lifecycle states, you need an AWS Identity and Access Management role and policies for locating the new software bill of materials in Amazon S3 and updating the package version in AWS IoT Core.

First, you will place the updated software bill of materials in your versioned Amazon S3 bucket and call the [UpdatePackageVersion] API operation with the sboms parameter included. Next, your authorized principal will assume the IAM role you created, locate the updated software bill of materials in Amazon S3, and update the package verison in AWS IoT Core for Software Package Catalog.

The following policies are required to perform this update:

Policies

- **Trust policy** Policy establishing a trust relationship with the authorized principal assuming the IAM role so it can locate the updated software bill of materials from your versioned bucket in Amazon S3 and update the package version in AWS IoT Core.

  -
    ```
    {
        "Version":"2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
                    "Service": "s3.amazonaws.com"
            },
                "Action": "sts:AssumeRole"
            }
        ]
    }
    ```
  -
    ```
    {
        "Version":"2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
                    "Service": "iot.amazonaws.com"
            },
                "Action": "sts:AssumeRole"
            }
        ]
    }
    ```

- **Permissions policy**: Policy to access the Amazon S3 versioned bucket where the software bill of materials are stored for a package version and update the package version in AWS IoT Core.

  -
    ```
    {
        "Version":"2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "s3:GetObject"
                ],
                "Resource": [
                    "arn:aws:s3:::awsexamplebucket1"
                ]
            }
        ]
    }
    ```
  -
    ```
    {
        "Version":"2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "iot:UpdatePackageVersion"
                ],
                "Resource": [
                    "arn:aws:iot:*:111122223333:package/<packageName>/
    version/<versionName>"
                ]
            }
        ]
    }
    ```

- **Pass role permissions**: Policy granting permission to pass the IAM role to Amazon S3 and AWS IoT Core when you call the UpdatePackageVersion API operation.

  -
    ```
    {
      "Version":"2012-10-17",
      "Statement": [
        {
    ```

```
            "Effect": "Allow",
            "Action": [
                "iam:PassRole",
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::awsexamplebucket1"
            ]
        }
    ]
}
```

-
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole",
                "iot:UpdatePackageVersion"
            ],
            "Resource": "arn:aws:iam::111122223333:role/<roleName>"
        }
    ]
}
```

Note

You can't update the software bill of materials on a package version that has transitioned to the `Deleted` lifecycle state.

For more information on creating an IAM role for an AWS service, see Creating a role to delegate permission to an AWS service.

For more information on creating an Amazon S3 bucket and uploading objects to it, see Creating a bucket and Uploading objects.

Preparing fleet indexing

With AWS IoT fleet indexing, you can search and aggregate data by using the reserved named shadow (`$package`). You can also group AWS IoT things by querying the Reserved named shadow and dynamic thing groups. For example, you can find information about which AWS IoT things use a specific package version, don't have a specific package version installed, or don't have any package version installed. You can gain further insight by combining attributes. For example, identifying things that have a specific version and are of a specific thing type (such as version 1.0.0 and thing type of pump_sensor). For more information, see Fleet indexing.

Setting the $package shadow as a data source

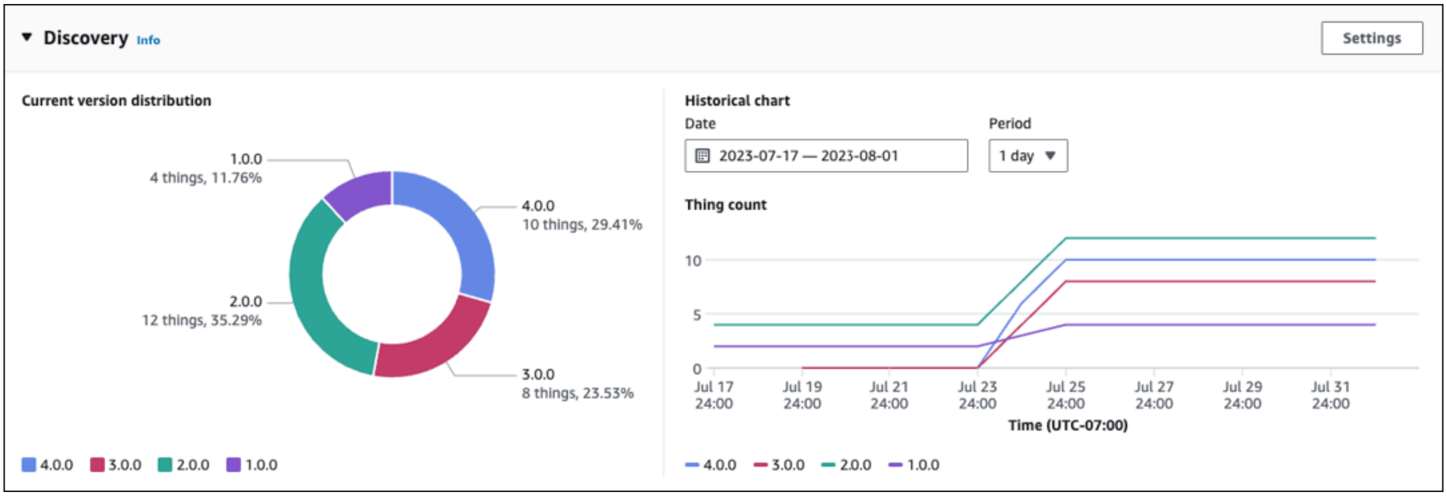
To use fleet indexing with Software Package Catalog, you must enable fleet indexing, set the named shadow as the data source, and define `$package` as the named shadow filter. If you haven't enabled fleet indexing, you can enable it within this process. From AWS IoT Core in the console, open **Settings**, choose **Manage indexing**, then **Add named shadows**, **Add device software packages and versions**, and **Update**. For more information, see Manage thing indexing.

Alternately, you can enable fleet indexing when you create your first package. When the **Enable dependencies for package management** dialog box appears, choose the option to add device software packages and versions as data sources to fleet indexing. By selecting this option, you also enable fleet indexing.

Note

Enabling fleet indexing for Software Package Catalog incurs standard service costs. For more information, see AWS IoT Device Management, Pricing.

Metrics displayed in the console



On the AWS IoT console software package details page, the **Discovery** panel displays standard metrics ingested through the `$package` shadow.

- The **Current version distribution** chart shows the number of devices and percentage for the 10 most recent package versions that are associated to an AWS IoT thing from all the devices associated to this software package. **Note:** If the software package has more package versions than those labeled in the chart, you can find them grouped within **Other**.
- The **Historical chart** shows the number of devices associated with selected package versions over a specified time period. The chart is initially empty until you select up to 5 package versions and define the date range and time interval. To select the chart's parameters, choose **Settings**. The data displayed in the **Historical chart** might be different than the **Current version distribution** chart because of the difference in number of package versions that they display and also because you can choose which package versions to analyze in the **Historical chart**. **Note:** When you select a package version to visualize, it counts toward the maximum number of fleet metrics limits. For more information, see [Fleet indexing limits and quotas](#).

For another method to gain insight into collecting package version distribution, see [Collecting package version distribution through `getBucketsAggregation`](#).

Query patterns

Fleet indexing with Software Package Catalog uses most standard supported features (such as terms, phrases, and search fields). However, comparison operators (for example, less than < and greater than >) and `range` queries are not available for the reserved named shadow (`$package`) version key. These queries are available for the attributes key. For more information, see Query syntax.

Example data

Note: for information about the reserved named shadow and its structure, see Reserved named shadow.

In this example, a first device is named AnyThing and has the following packages installed:

- Software package: SamplePackage

  Package version: 1.0.0

  Package ID: 1111

The shadow looks as follows:

```
{
    "state": {
        "reported": {
            "SamplePackage": {
                "version": "1.0.0",
                "attributes": {
                    "s3UrlForSamplePackage": "https://EXAMPIEBUCKET.s3.us-
west-2.amazonaws.com/exampleCodeFile1",
                    "packageID": "1111"
                    }
            }
        }
    }
}
```

A second device is named AnotherThing and has the following package installed:

- Software package: SamplePackage

  Package version: 1.0.0

  Package ID: 1111
- Software package: OtherPackage

Package version: 1.2.5

Package ID: 2222

The shadow looks as follows:

```
{
    "state": {
        "reported": {
            "SamplePackage": {
                "version": "1.0.0",
                "attributes": {
                    "s3UrlForSamplePackage": "https://EXAMPIEBUCKET.s3.us-
west-2.amazonaws.com/exampleCodeFile1",
                    "packageID": "1111"
                }
            },
            "OtherPackage": {
                "version": "1.2.5",
                "attributes": {
                    "s3UrlForOtherPackage": "https://EXAMPIEBUCKET.s3.us-
west-2.amazonaws.com/exampleCodeFile2",
                    "packageID": "2222"
                    }
            },
        }
    }
}
```

Sample queries

The following table lists sample queries based on the example device shadows for `AnyThing` and `AnotherThing`. For more information, see Example thing queries.

Latest version of AWS IoT Device Tester for FreeRTOS

Requested information	Query	Result
Things that have a specific package version installed	`shadow.name.$package.reported.Sample`	`AnyThing`, `OtherThing`

Requested information	Query	Result
	`Package.version:1.0.0`	
Things that don't have a specific package version installed	`NOT shadow.name.$package.reported.OtherPackage.version:1.2.5`	`AnyThing`
Any device using a package version whose package ID is greater than 1500	`shadow.name.$package.reported.*.attributes.packageID>1500"`	`OtherThing`
Things that have a specific package installed and have more than one package installed	`shadow.name.$package.reported.SamplePackage.version:1.0.0 AND shadow.name.$package.reported.totalCount:2`	`OtherThing`

Collecting package version distribution through `getBucketsAggregation`

In addition to the **Discovery** panel within the AWS IoT console, you can also get package version distribution information by using the [GetBucketsAggregation](#) API operation. To get the package version distribution information, you must do the following:

- Define a custom field within fleet indexing for each software package. **Note:** Creating custom fields count toward [AWS IoT fleet indexing service quotas](#).
- Format the custom field as follows:

```
shadow.name.$package.reported.<packageName>.version
```

For more information, see the [Custom fields](#) section in AWS IoT fleet indexing.

Preparing AWS IoT Jobs

AWS IoT Device Management Software Package Catalog extends AWS IoT Jobs through substitution parameters, and integration with AWS IoT fleet indexing, dynamic thing groups, and the AWS IoT thing's reserved named shadow.

> **Note**
>
> To use all the functionality that Software Package Catalog offers, you must create these AWS Identity and Access Management (IAM) roles and policies: AWS IoT Jobs rights to deploy package versions and AWS IoT Jobs rights to update the reserved named shadow. For more information, see Preparing security.

Substitution parameters for AWS IoT jobs

You can use substitution parameters as a placeholder within your AWS IoT job document. When the job service encounters a substitution parameter, it points the job to a named software version's attribute for the parameter value. You can use this process to create a single job document and pass the metadata into the job through general-purpose attributes. For example, you might pass an Amazon Simple Storage Service(Amazon S3) URL, a software package Amazon Resource Name (ARN), or a signature into the job document through package version attributes.

The substitution parameters should be formatted in the job document as follows:

- **Software Package Name and Package Version**
  - The empty string between `package::version` represents the software package name substitution parameter. The empty string between `version::attribute` represents the software package version substitution parameter. Refer to the following example for using the package name and package verion substituion parameters in a job document: `${aws:iot:package::version::attributes:<attributekey>}`.
  - The job document will autofill these substituion parameters using the *Version ARN* from the package version details. If you're creating a job or job template for a single-package deployment using an API or CLI command, the *Version ARN* for a package version is represented by the `destinationPackageVersions` parameter in `CreateJob` and `DescribeJob`.
- **All Attributes for a Software Package Version**

- Refer to the following example for using the all attributes of a software package version substitution parameter in a job document: `${aws:iot:package:<packageName>:version:<versionName>:attributes}`

> **Note**
>
> The package name, package version, and all attributes substitution parameters can be used together. Refer to the following example for using all three substitution parameters in a job document: `${aws:iot:package::version::attributes}`

In the following example, there is a software package named `samplePackage` and it has a package version named `2.1.5` that has the following attributes:

- name: `s3URL`, value: `https://EXAMPIEBUCKET.s3.us-west-2.amazonaws.com/exampleCodeFile`
  - This attribute identifies the location of the code file that's stored within Amazon S3.
- name: `signature`, value: `aaaaabbbbbcccccdddddeeeeefffffggggghhhhhiiiiijjjjj`
  - This attribute provides a code signature value that the device requires as a security measure. For more information, see Code Signing for jobs. **Note:** This attribute is an example and not required as part of Software Package Catalog or jobs.

For `s3URL`, the job document parameter is written as follows:

```
{
"samplePackage": "${aws:iot:package:samplePackage1:version:2.1.5:attributes:s3URL}"
}
```

For `signature`, the job document parameter is written as follows:

```
{
"samplePackage": "${aws:iot:package:samplePackage1:version:2.1.5:attributes:signature}"
}
```

The complete job document is written as follows:

```
{
```

```
  ...
  "Steps": {
    "uninstall": ["samplePackage"],
    "download": [
      {
        "samplePackage":
 "${aws:iot:package:samplePackage1:version:2.1.5:attributes:s3URL}"
      },
    ],
    "signature": [
      "samplePackage" :
 "${aws:iot:package:samplePackage1:version:2.1.5:attributes:signature}"
    ]
  }
}
```

After the substitution is made, the following job document is deployed to the devices:

```
{
  ...
  "Steps": {
    "uninstall": ["samplePackage"],
    "download": [
      {
        "samplePackage": "https://EXAMPIEBUCKET.s3.us-west-2.amazonaws.com/
exampleCodeFile"
      },
    ],
    "signature": [
      "samplePackage" : "aaaaabbbbbcccccdddddeeeeefffffggggghhhhhiiiiijjjjj"
    ]
  }
}
```

Substitution Parameters (Before and After View)

Substitution parameters streamline the creation of a job document using various flags such as `$default` for the default package version. This eliminates the need to manually enter specific package version metadata for each job deployment as those flags are autofilled with the referenced metadata in the specific package verison. For more information on package version attributes such as `$default` for the default package version, see Preparing the job document and package version for deployment.

In the AWS Management Console, toggle the *Preview substitution* button in the *Deployment instruction file editor* window during a job deployment for a package version to view the job document with and without the substitution parameters.

Using the "before-substitution" parameter in the `DescribeJob` and `GetJobDocument` APIs, you can view the API response before and after the substitution parameters are removed. Refer to the following examples with the `DescribeJob` and `GetJobDocument` APIs:

- `DescribeJob`
  - Default view

```
{
    "jobId": "<jobId>",
    "description": "<description>",
    "destinationPackageVersions": ["arn:aws:iot:us-west-2:123456789012:package/
TestPackage/version/1.0.2"]
}
```

  - Before substituion view

```
{
    "jobId": "<jobId>",
    "description": "<description>",
    "destinationPackageVersions": ["arn:aws:iot:us-west-2:123456789012:package/
TestPackage/version/$default"]
}
```

- `GetJobDocument`
  - Default view

```
{
    "attributes": {
        "location": "prod-artifacts.s3.us-east-1.amazonaws.com/mqtt-core",
        "signature": "IQoJb3JpZ2luX2VjEIrwEaCXVzLWVhc3QtMSJHMEUCIAofPNPpZ9cI",
        "streamName": "mqtt-core",
        "fileId": "0"
    },
}
```

  - Before substituion view

```
{
    "attributes": "${aws:iot:package:TestPackage:version:$default:attributes}",
}
```

For more information about AWS IoT Jobs, creating job documents, and deploying jobs, see Jobs.

Preparing the job document and package version for deployment

When a package version is created, it's in a `draft` state to indicate that it's being prepared for deployment. To prepare the package version for deployment, you must create a job document, save the document in a location that the job can access (such as Amazon S3), and confirm that the package version has the attribute values that you want the job document to use. (Note: You can update attributes for a package version only while it's in the `draft` state.)

When you create an AWS IoT Job or Job template for a single-package deployment, you have the following options to customize your job document:

Deployment instruction file (`recipe`)

- The deployment instruction file for a package version contains the deployment instructions, including an inline job document, for deploying a package version to multiple devices. The file associates specific deployment instructions to a package version for a quick and efficient job deployment.

  In the AWS Management Console, you can create the file in the *Deployment instructions file preview* window in the *Version deployment configurations* tab of the create new package workflow. You can leverage AWS IoT to automatically generate an instruction file from your package version attributes using *Start from AWS IoT recommended file* or use your existing job document stored in an Amazon S3 bucket using *Use your own deployment instruction file*.

  > **Note**
  >
  > If you use your own job document, you can update it directly in the *Deployment instructions file preview* window, but it will not automatically update your original job document stored in your Amazon S3 bucket.

When using the AWS CLI or an API command such as `CreatePackageVersion`, `GetPackageVersion`, or `UpdatePackageVersion`, `recipe` represents the deployment instruction file, which includes an inline job document.

For more information on what a job document is, see [Basic concepts](#).

Refer to the following example for the deployment instruction file as represented by `recipe`:

```
{
    "packageName": "sample-package-name",
    "versionName": "sample-package-version",
    ...
    "recipe": "{...}"
}
```

> **Note**
>
> The deployment instruction file as represented by `recipe` can be updated when a package version is in the `published` status state as it's separate from the package version metadata. It become immutable during job deployment.

`Artifact` version attribute

- Using the version attribute `artifact` in your software package version, you can add the Amazon S3 location for your package version artifacts. When a job deployment for your package version is triggered using AWS IoT Jobs, the presigned URL placeholder ${aws:iot:package:<*packageName*>:version:<*versionName*>:artifact-location:*s3-presigned-url*} in the job document will be updated using the Amazon S3 bucket, bucket key, and version of the file stored in the Amazon S3 bucket. The Amazon S3 bucket storing the package version artifacts must be located in the same region where the package version was created.

> **Note**
>
> To store multiple object versions of the same file in your Amazon S3 bucket, you must enable versioning on your bucket. For more information, see [Enabling versioning on buckets](#).

To access the package version artifacts in the Amazon S3 bucket when using the `CreatePackageVersion` or `UpdatePackageVersion` API operation, you must have the following permissions:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:GetObjectVersion",
            "Resource": "arn:aws:s3:::bucket-name/key-name"
        }
    ]
}
```

For more information on the version attribute `artifact` in the `CreatePackageVersion` and `UpdatePackageVersion` API operations, see [CreatePackageVersion](#) and [UpdatePackageVersion](#).

Refer to the following example that shows the version attribute `artifact` supporting the artifact location in Amazon S3 when creating a new package version:

```
{
    "packageName": "sample package name",
    "versionName": "1.0",
    "artifact": {
        "s3Location": {
            "bucket": "firmware",
            "key": "image.bin",
            "version": "12345"
        }
    }
```

```
}
```

> **Note**
>
> When a package version updates from a `draft` status state to a `published` status state, the package version attributes and artificats location become immutable. To update this informaiton, you need to create a new package version and perform those updates while in the `draft` status state.

Package Version

- A default software package version can be denoted in the available versions of the software package providing a secure and stable package version. This serves as the baseline version of the software package when deploying the default package version to your device fleet using AWS IoT Jobs. When creating a job to deploy the `$default` package version for a software package, the package version in the job document and in the new job deployment must match as `$default`. The package version in the job deployment is represented by `destinationPackageVersions` for API and CLI commands and `VersionARN` in the AWS Management Console. The package version in the job document is represented by the following job document placeholder shown below:

  ```
  arn:aws:iot:<regionCode>:111122223333:package/<packageName>/version/$default
  ```

  To create a job or job template using the default package version, use the `$default` flag in the `CreateJob` or `CreateJobTemplate` API command as shown below:

  ```
  "$ aws iot create-job \
      --destination-package-versions "arn:aws:iot:us-west-2:123456789012:package/
  TestPackage/version/$default"
      --document file://jobdoc.json
  ```

  > **Note**
  >
  > The `$default` package version attribute referencing the default version is an optional attribute that is only required when referencing the default package version for a job deployment via AWS IoT Jobs.

When you are satisfied with the package version, publish it either through the software package details page in the AWS IoT console or by issuing the UpdatePackageVersion API operation. You can then reference the package version when you create the job either through the AWS IoT console or by issuing the CreateJob API operation.

Naming the packages and versions when deploying

To deploy a software package version to a device, confirm the software package and package version referenced in the job document match the software package and package version stated in the `destinationPackageVersions` parameter in the `CreateJob` API operation. If they don't match, you will receive an error message prompting you to make both references match. For more information on Software Package Catalog error messages, see General Troubleshooting Error Messages.

In addition to the software packages and package versions referenced in the job document, you can include additional software packages and package versions in the `destinationPackageVersions` parameter in the `CreateJob` API operation not referenced in the job document. Ensure the necessary installation information is included in the job document for devices to properly install the additional software package versions. For more information on the `CreateJob` API operation, see CreateJob.

Targeting jobs through AWS IoT dynamic thing groups

Software Package Catalog works with fleet indexing, AWS IoT jobs, and AWS IoT dynamic thing groups to filter and target devices within your fleet to select which package version to deploy to your devices. You can run a fleet indexing query based on your device's current package information and target those things for an AWS IoT job. You can also release software updates, but only to eligible target devices. For example, you can specify that you want to deploy a configuration only to those devices that currently run the `iot-device-client 1.5.09`. For more information, see Create a dynamic thing group.

Reserved named shadow and package versions

If configured, AWS IoT Jobs can update a thing’s reserved named shadow (`$package`)when the job successfully completes. If you do so, you don’t need to manually associate a package version to a thing’s reserved named shadow.

You might choose to manually associate or update a package version to the thing’s reserved named shadow in the following situations:

- You register a thing to AWS IoT Core without associating the installed package version.
- AWS IoT Jobs isn't configured to update the thing's reserved named shadow.
- You use an in-house process to dispatch package versions to your fleet and that process doesn't update AWS IoT Core when it completes.

> **Note**
>
> We recommend you use AWS IoT Jobs to update the package version in the reserved named shadow (`$package`). Updating the version parameter in the `$package` shadow through other processes (such as, manual or programmatic API calls) when AWS IoT Jobs is also configured to update the shadow, can cause inconsistencies between the actual version on device and version reported to the reserved named shadow.

You can add or update a package version to a thing's reserved named shadow (`$package`) through the console or the [`UpdateThingShadow`] API operation. For more information, see [Associating a package version to an AWS IoT thing].

> **Note**
>
> Associating a package version to an AWS IoT thing doesn't directly update the device software. You must deploy the package version to the device to update the device software.

Uninstalling a software package and its package version

`$null` is a reserved placeholder that prompts the AWS IoT Jobs service to remove the existing software package and package version from the device's reserved named shadow `$package`. For more information, see [Reserved named shadow.]

To use this feature, replace the version name at the end of the [destinationPackageVersion] Amazon Resource Name (ARN) with `$null`. Afterward, you must instruct your service to remove the software from the device.

The authorized ARN uses the following format:

```
arn:aws:iot:<regionCode>:111122223333:package/<packageName>/version/$null
```

For example,

```
$ aws iot create-job \
    ... \
    --destinationPackageVersions ["arn:aws:iot:us-east-1:111122223333:package/
samplePackage/version/$null"]
```

Getting started with Software Package Catalog

You can build and maintain the AWS IoT Device Management Software Package Catalog through the AWS Management Console, AWS IoT Core API operations, and AWS Command Line Interface (AWS CLI).

> **Note**
>
> Enabling AWS IoT fleet indexing is a requirement to use Software Package Catalog. Basic operations such as creating a software package version in the AWS Management Console and using the `CreatePackage` API command will fail without AWS IoT fleet indexing enabled.
> For more information on using AWS IoT fleet indexing with Software Package Catalog, see Preparing fleet indexing.

Using the console

To use the AWS Management Console, sign into your AWS account and navigate to AWS IoT Core. In the navigation pane, choose **Software packages**. You can then create and manage packages and their versions from this section.

Using API or CLI operations

You can use the AWS IoT Core API operations to create and manage Software Package Catalog features. For more information, see AWS IoT API Reference and AWS SDKs and Toolkits. The AWS CLI commands also manage your catalog. For more information, see the AWS IoT CLI Command Reference.

This chapter contains the following sections:

Creating a software package and package version

You can use the following steps to create a package and an initial version thing through the AWS Management Console.

To create a software package

1. Sign into your AWS account and navigate to the AWS IoT console.
2. On the navigation pane, choose **Software packages**.
3. On the **AWS IoT software package** page, choose **Create package**. The **Enable dependencies for package management** dialog box appears.
4. Under **Fleet indexing**, select **Add device software packages and version**. This is required for Software Package Catalog and provides fleet indexing and metrics about your fleet.
5. [Optional] If you want AWS IoT jobs to update the reserved named shadow when jobs successfully complete, select **Auto update shadows from jobs**. If you do not want AWS IoT jobs to make this update, leave this check-box unselected.
6. [Optional] To grant AWS IoT jobs the rights to update the reserved named shadow, under **Select role**, choose **Create role**. If you don't want AWS IoT jobs to make this update, this role is not required.
7. Create or select a role.

   a. If you **don’t have a role** for this purpose: When the **Create role** dialog box appears, enter a **Role name**, and then choose **Create**.

   b. If you **do have a role** for this purpose: For **Select role**, choose your role and then make sure the **Attach policy to IAM role** check box is selected.
8. Choose **Confirm**. The **Create new package** page appears.
9. Under **Package detail**, enter a **Package name**.
10. Under **Package description**, enter information to help you identify and manage this package.
11. [Optional] You can use tags to help you categorize and manage this package. To add tags, expand **Tags**, choose **Add tag**, and enter a key-value pair. You can enter up to 50 tags. For more information, see Tagging your AWS IoT resources.

To add a package version while creating a new package

1. Under **Initial version**, enter a **Version name**.

   We recommend using the SemVer format (for example, `1.0.0.0`) to uniquely identify your package version. You are also able to use a different formatting strategy that better suits your use case. For more information, see Package version lifecycle.

2. Under **Version description**, enter information that will help you identify and manage this package version .

   > **Note**
   >
   > The **Default version** check box is deactivated because package versions are created in a `draft` state. You can name the default version after you create the package version and when you change the state to `published`. For more information, see Package version lifecycle.

3. [Optional] To help you manage this version or to communicate information to your devices, enter one or more name-value pairs for **Version attributes**. Choose **Add attribute** for each name-value pair you enter. For more information, see Version attributes.

4. [Optional] You can use tags to help you categorize and manage this package. To add tags, expand **Tags**, choose **Add** tag, and enter a key-value pair. You can enter up to 50 tags. For more information, see Tagging your AWS IoT resources.

5. Choose **Next**.

Associate the Software Bill of Materials to a Package Version (Optional)

1. On **Step 3: Version SBOMs (Optional)** in the **SBOM configurations** window, choose the default SBOM file format and validation mode used to validate your software bill of materials before it is associated to your package version.

2. In the **Add SBOM file** window, enter the Amazon Resource Name (ARN) representing your versioned Amazon S3 bucket and the preferred SBOM file format if the default type doesn't work.

> **Note**
>
> You can either add a single SBOM file or a single zip file containing multiple SBOMs if you have more than one software bill of material for your package version.

3. In the **Added SBOM file** window, you can view the SBOM file you added for your package version.

4. Choose **Create package and version**. The package version page appears and you can see the validation status of your SBOM file in the **Added SBOM file** window. The initial status will be `In progress` as the SBOM file undergoes validation.

> **Note**
>
> The SBOM file validation statuses are `Invalid file`, `Not started`, `In progress`, `Validated (SPDX)`, `Validated (CycloneDX)`, and the validation failure reasons.

Deploying a package version through AWS IoT jobs

You can use the following steps to deploy a package version through the AWS Management Console.

Prerequisites:

Before you begin, do the following:

- Register AWS IoT things with AWS IoT Core. For directions to add your devices to AWS IoT Core, see Create a thing object.
- [Optional] Create an AWS IoT thing group or dynamic thing group to target the devices that you will deploy the package version. For directions to create a thing group, see Create a static thing group. For directions to create a dynamic thing group, see Create a dynamic thing group.
- Create a software package and a package version. For more information, see Creating a software package and package version.
- Create a job document. For more information, see Preparing the job document and package version for deployment.

To deploy an AWS IoT job

1. On the [AWS IoT console](), choose **Software packages**.
2. Choose the software package that you want to deploy. The **software package details** page appears.
3. Choose the package version that you want to deploy, under **Versions**, and choose **Deploy job version**.
4. If this is your first time deploying a job through this portal, a dialog box describing the requirements appears. Review the information and choose **Acknowledge**.
5. Enter a name for the deployment or leave the autogenerated name in the **Name** field.
6. [Optional] In the **Description** field, enter a description that identifies the purpose or contents of the deployment, or leave the autogenerated information.

   Note: We recommend that you don't use personally identifiable information in the Job name and description fields.
7. [Optional] Add any tags to associate with this job.
8. Choose **Next**.
9. Under **Job targets**, choose the things or thing groups that should receive the job.
10. In the **Job file** field, specify the job document JSON file.
11. Open **Jobs integration with the Package Catalog service**.
12. Select the packages and versions that are specified within your job document.

> **Note**
>
> You are required to choose the same packages and package versions that are specified within the job document. You can include more, but the job will issue instructions only for the packages and versions included in the job document. For more information, see [Naming the packages and versions when deploying]().

13. Choose **Next**.
14. On the Job configuration page, select one of the following job types in the Job configuration dialog box:

    - **Snapshot job:** A snapshot job is complete when it's finished its run on the target devices and groups.

- **Continuous job:** A continuous job applies to thing groups and runs on any device that you later add to a specified target group.

15. In the **Additional configurations - optional** dialog box, review the following optional job configurations and make your selections accordingly. For more information, see Job rollout, scheduling, and abort configurations and Job execution timeout and retry configurations.

    - Rollout configuration
    - Scheduling configuration
    - Job executions timeout configuration
    - Job executions retry configuration
    - Abort configuration

16. Review the job selections and then choose **Submit**.

After you create the job, the console generates a JSON signature and places it in your job document. You can use the AWS IoT console to view the status of a job, or cancel or delete a job. To manage jobs, go to the Job hub of the console.

Associating a package version to an AWS IoT thing

After you install software on your device, you can associate a package version to an AWS IoT thing's reserved named shadow. If AWS IoT jobs has been configured to update the thing's reserved named shadow after the job deploys and successfully completes, you don't need to complete this procedure. For more information, see Reserved named shadow.

Prerequisites:

Before you begin, do the following:

- Create an AWS IoT thing, or things, and establish telemetry through AWS IoT Core. For more information, see Getting started with AWS IoT Core.
- Create a software package and package version. For more information, see Creating a software package and package version.
- Install the package version software on the device.

> **Note**
>
> Associating a package version to an AWS IoT thing doesn't update or install software on the physical device. The package version must be deployed to the device.

To associate a package version to an AWS IoT thing

1. On the AWS IoT console navigation pane, expand the **All devices** menu and choose **Things**.
2. Identify the AWS IoT thing that you want to update from the list and choose the thing name to display its details page.
3. In the **Details** section, choose **Packages and versions**.
4. Choose **Add to package and version**.
5. For **Choose a device package**, choose the software package you want.
6. For **Choose a version**, choose the software version you want.
7. Choose **Add device package**.

   The package and version appear on the **Selected packages and versions** list.
8. Repeat these steps for each package and version that you want to associate to this thing.
9. When you're finished, choose **Add package and version details**. The **Thing details** page opens and you can see the new package and version in the list.

AWS IoT Jobs

Use AWS IoT Jobs to define a set of remote operations that can be sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install applications, run firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

Accessing AWS IoT jobs

You can get started with AWS IoT Jobs by using the console or the AWS IoT Core API.

Using the console

Sign in to the AWS Management Console, and go to the AWS IoT console. In the navigation pane, choose **Manage**, and then choose **Jobs**. You can create and manage jobs from this section. If you want to create and manage job templates, in the navigation pane, choose **Job templates**. For more information, see Create and manage jobs by using the AWS Management Console.

Using the API or CLI

You can get started by using the AWS IoT Core API operations. For more information, see AWS IoT API Reference. The AWS IoT Core API that AWS IoT jobs is built on is supported by the AWS SDK. For more information, see AWS SDKs and Toolkits.

You can use the AWS CLI to run commands for creating and managing jobs and job templates. For more information, see AWS IoT CLI reference.

AWS IoT Jobs Regions and endpoints

AWS IoT Jobs supports control plane and data plane API endpoints that are specific to your AWS Region. The data plane API endpoints are specific to your AWS account and AWS Region. For more information about the AWS IoT Jobs endpoints, see AWS IoT Device Management - jobs data endpoints in the *AWS General Reference*.

What is a remote operation?

> **Note**
>
> Over-the-air (OTA) updates are not available in the following regions:

- Europe (Spain)
- Asia Pacific (Malaysia)

A remote operation is any update or action you can perform on a physical device, virtual device, or endpoint that can be done remotely without the need for the physical presence of an operator or technician. The remote operation is performed using an over-the-air (OTA) update so your devices don't have to be physically present. Managing your device fleet in the AWS Cloud allows you to perform remote operations on your devices when they are registered with AWS IoT Core.

AWS IoT Device Management Jobs offers a scalable approach for performing remote actions on your devices registered with AWS IoT Core. A job is created in the AWS Cloud and pushed out to all targeted devices using an OTA update via the MQTT or HTTP protocol.

AWS IoT Device Management Jobs provide you the capability to perform remote operations such as factory resets, device reboots, and software OTA updates in a secure, scalable, and more cost-effective way.

For more information on AWS IoT Core, see What is AWS IoT?.

For more information on AWS IoT Device Management Jobs, see What is AWS IoT Jobs?.

Benefits of using AWS IoT Device Management Jobs for remote operations

Using AWS IoT Device Management Jobs to perform your remote operations streamlines the management of your device fleet. The following list highlights some of the key benefits for using AWS IoT Device Management Jobs to perform your remote operations:

- **Seamless integration with other AWS services**
  - AWS IoT Device Management Jobs integrates closely with the following value-added AWS services and features:
    - **Amazon S3**: Store your remote operation instructions in a secure Amazon S3 bucket where you control the access permissions for that content. Using an Amazon S3 bucket provides a scalable and durable storage solution that natively intergrates with AWS IoT Device Management Software Package Catalog allowing AWS IoT Device Management Jobs to reference and substitute in update instructions. For more information, see What is Amazon S3?.

- **Amazon CloudWatch**: Monitor and log the remote operation implementation status of the job execution for each device in addition to other device activity to track and analyze the overall job performance in AWS IoT Device Management Jobs. For more information, see What is Amazon CloudWatch? Monitoring jobs logs and capturing historical data for troubleshooting. How it works with jobs.
- **AWS IoT Device Shadow service**: Maintain a digital representation of your AWS IoT thing via a device shadow using AWS IoT Device Management Jobs so your device's state is available to applications and other services regardless of device connectivity. For more information, see AWS IoT Device Shadow service.

- **Security best practices**
  - **Permission control**: Control the access permissions to your remote operating instructions using Amazon S3 and determine which IAM users can deploy your remote operating instructions to your device fleet using AWS IoT policies and IAM user roles.
    - For more information on AWS IoT policies, see Create an AWS IoT policy.
    - For more information on IAM user roles, see Identity and access management for AWS IoT.
- **Scalability**
  - **Targeted job deployment**: Control which devices receive the job document from a job with a targeted job deployment using specific device grouping criteria entered in your job document when creating the job. Creating an AWS IoT thing for each device and storing that information in the AWS IoT registry allows you to perform targeted searches using fleet indexing. You can create custom groups based on the fleet indexing search results to support your target job deployment. For more information, see Managing devices with AWS IoT. Use jobs to do snapshot vs continuous jobs.
  - **Job status**: Track the status of the job document rollout to your device fleet and overall job status from a device fleet level in addition to the individual implementation status of the job document on each device. For more information, see Jobs and job execution states.
  - **New device scalability**: Easily deploy your job document to a new device by adding it to an existing, custom group created using fleet indexing via a continuous job. This will save you time over having to deploy the job document to each new device separately. Or, you can use a more targeted approach with a snapshot shot by deploying a job document to a predetermined group of devices once and then the job is completed.
- **Flexibility**

- **Job configurations**: Customize your job and job document with the optional job configurations rollout, scheduling, abort, timeout, and retry to meet your specific needs. For more information, see Job configurations.
- **Cost effective**
  - Introduce a more efficient cost structure for maintaining your device fleet by leveraging AWS IoT Device Management Jobs to deploy critical updates and perform routine maintenance tasks. A do-it-yourself (DIY) solution to maintain your device fleet includes recurring, variable costs such as infrastructure required to host and manage the DIY solution, labor costs to develop, maintain, and scale the DIY solution, and data transmission costs. Leveraging the transparent, fixed cost structure of AWS IoT Device Management Jobs, you know exactly what each job execution for a device will cost in addition to the data transmission costs required to facilitate the job document rollout to your device fleet and tracking the job execution status for each device. For more information, see AWS IoT Core pricing.

What is AWS IoT Jobs?

Use AWS IoT Jobs to define a set of remote operations that can be sent to and run on one or more devices connected to AWS IoT.

To create jobs, first define a *job document* that contains a list of instructions describing operations that the device must perform remotely. To perform these operations, specify a list of *targets*, which are individual things, thing groups, or both. The job document and targets together constitute a *deployment*.

Each deployment can have additional configurations:

- **Rollout**: This configuration defines how many devices receive the job document every minute.
- **Abort**: If a certain number of devices don't receive the job notification, use this configuration to cancel the job. This avoids sending a bad update to an entire fleet.
- **Timeout**: If a response isn't received from your job targets within a certain duration, the job can fail. You can track the job that's running on these devices.
- **Retry**: If a device reports failure or a job times out, you can use AWS IoT Jobs to resend the job document to the device automatically.
- **Scheduling**: This configuration enables you to schedule a job for a future date and time. It also enables you to create recurring maintenance windows that update devices during predefined, low-traffic periods.

AWS IoT Jobs sends a message to inform the targets that a job is available. The target starts the *execution* of the job by downloading the job document, performing the operations it specifies, and reporting its progress to AWS IoT. You can track the progress of a job for a specific target or for all targets by running commands that are provided by AWS IoT Jobs. When a job starts, it has a status of *In progress*. The devices then report incremental updates while displaying this status until the job succeeds, fails, or times out.

The following topics describe some key concepts of jobs and the lifecycle of jobs and job executions.

Topics

Jobs key concepts

The following concepts provide details about AWS IoT Jobs and how to create and deploy jobs to run remote operations on your devices.

Basic concepts

The following are basic concepts you must know when using AWS IoT Jobs.

Job

A job is a remote operation that is sent to and run on one or more devices connected to AWS IoT. For example, you can define a job that instructs a set of devices to download and install an application or run firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

Job document

To create a job, you must first create a job document that is a description of the remote operations to be performed by the devices.

Job documents are UTF-8 encoded JSON documents and contain information that your devices require to perform a job. A job document contains one or more URLs where the device can download an update or other data. The job document can be stored in an Amazon S3 bucket, or be included inline with the command that creates the job.

> **Note**
>
> In MQTT responses, the `jobDocument` field is a JSON object. In HTTP responses, it's a string representation of the JSON object.

Target

When you create a job, you specify a list of targets that are the devices that should perform the operations. The targets can be things or thing groups or both. The AWS IoT Jobs service sends a message to each target to inform it that a job is available.

Deployment

After you create a job by providing the job document and specifying your list of targets, the job document is then deployed to the remote target devices for which you want to perform the update. For snapshot jobs, the job will complete after deploying to the target devices. For continuous jobs, a job is deployed to a group of devices as they are added to the groups.

Job execution

A job execution is an instance of a job on a target device. The target starts an execution of a job by downloading the job document. It then performs the operations specified in the document, and reports its progress to AWS IoT. An execution number is a unique identifier of a job execution on a specific target. The AWS IoT Jobs service provides commands to track the progress of a job execution on a target and the progress of a job across all targets.

Job types concepts

The following concepts can help you understand more about the different types of jobs that you can create with AWS IoT Jobs.

Snapshot job

By default, a job is sent to all targets that you specify when you create the job. After those targets complete the job (or report that they're unable to do so), the job is complete.

Continuous job

A continuous job is sent to all targets that you specify when you create the job. It continues to run and is sent to any new devices (things) that are added to the target group. For example, a continuous job can be used to onboard or upgrade devices as they're added to a group. You can make a job continuous by setting an optional parameter when you create the job.

> **Note**
>
> When targeting your IoT fleet using dynamic thing groups, we recommend that you use continuous jobs instead of snapshot jobs. By using continuous jobs, devices that join the group receive the job execution even after the job has been created.

Presigned URLs

For secure, time-limited access to data that's not included in the job document, you can use presigned Amazon S3 URLs. Place your data in an Amazon S3 bucket and add a placeholder link to the data in the job document. When AWS IoT Jobs receives a request for the job document, it parses the job document by looking for the placeholder links, and then replaces the links with presigned Amazon S3 URLs.

The placeholder link is in the following format:

```
${aws:iot:s3-presigned-url:https://s3.amazonaws.com/bucket/key}
```

where *bucket* is your bucket name and *key* is the object in the bucket to which you are linking.

In the Beijing and Ningxia Regions, presigned URLs work only if the resource owner has an ICP (Internet Content Provider) license. For more information, see Amazon Simple Storage Service in the *Getting Started with AWS Services in China* documentation.

Job configuration concepts

The following concepts can help you understand how to configure jobs.

Rollouts

You can specify how quickly targets are notified of a pending job execution. This allows you to create a staged rollout to better manage updates, reboots, and other operations. You can create a rollout configuration by using either a static rollout rate or an exponential rollout rate. To specify the maximum number of job targets to inform per minute, use a static rollout rate.

For examples of setting rollout rates and for more information about configuring job rollouts, see Job rollout, scheduling, and abort configurations.

Scheduling

Job scheduling enables you to schedule the rollout timeframe of a job document to all devices in the target group for continuous and snapshot jobs. Additionally, you can create an optional maintenance window containing specific dates and times that a job will rollout the job document to all devices in the target group. A maintenance window is a recurring instance with a frequency of daily, weekly, monthly, or custom dates and times selected during the initial job or job template creation. Only continuous jobs can be scheduled to perform a rollout during a maintenance window.

Jobs Scheduling is specific to your job. Individual Job Executions can't be scheduled. For more information, see Job rollout, scheduling, and abort configurations.

Abort

You can create a set of conditions to cancel rollouts when criteria that you specify have been met. For more information, see Job rollout, scheduling, and abort configurations.

Timeouts

Job timeouts notify you whenever a job deployment gets stuck in the `IN_PROGRESS` state for an unexpectedly long period of time. There are two types of timers: in-progress timers and step timers. When the job is `IN_PROGRESS`, you can monitor and track the progress of your job deployment.

Rollouts and abort configurations are specific to your job, whereas the timeout configuration is specific to a job deployment. For more information, see Job execution timeout and retry configurations.

Retries

Job retries make it possible to retry the job execution when a job fails, times out, or both. You can have up to 10 attempted retries to execute the job. You can monitor and track the progress of your retry attempt and whether the job execution succeeded.

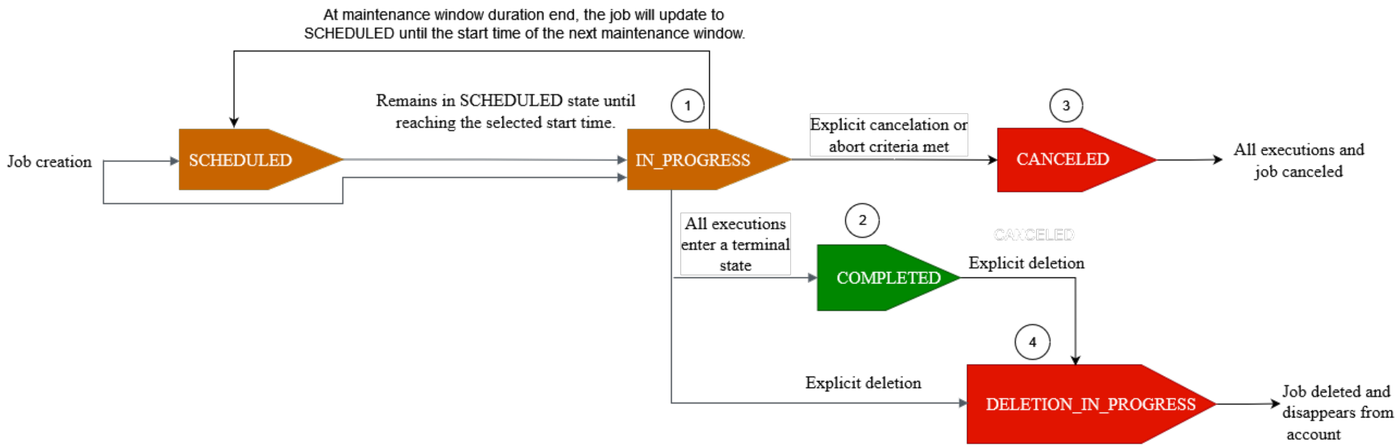
Rollouts and abort configurations are specific to your job, whereas the timeout and retry configurations are specific to a job execution. For more information, see Job execution timeout and retry configurations.

Jobs and job execution states

The following sections describe the lifecycle of an AWS IoT job and the lifecycle of a job execution.

Job states

The following diagram shows the different states of an AWS IoT job.



A job that you create using AWS IoT Jobs can be in one of the following states:

- **SCHEDULED**

  During the initial job or job template creation using the AWS IoT console, [CreateJob](#) API, or [CreateJobTemplate](#) API, you can select the optional scheduling configuration in the AWS IoT console or the `SchedulingConfig` in the [CreateJob](#) API or [CreateJobTemplate](#) API. When you start a scheduled job containing a specific `startTime`, `endTime`, and `endBehavior`, the job status updates to `SCHEDULED`. When the job reaches your selected `startTime` or the `startTime` of the next maintenance window (if you selected job rollout during a maintenance window), the status will update from `SCHEDULED` to `IN_PROGRESS` and begin rollout of the job document to all devices in the target group.

- **IN_PROGRESS**

  When you create a job using the AWS IoT console or the [CreateJob](#) API, the job status updates to `IN_PROGRESS`. During job creation, AWS IoT Jobs starts rolling out job executions to the devices in your target group. After all the job executions have rolled out, AWS IoT Jobs waits for devices to complete the remote action.

  For information about concurrency and limits that apply to in-progress jobs, see [AWS IoT Jobs limits](#).

> **Note**
>
> When an IN_PROGRESS job reaches the end of the current maintenance window, the rollout of the job document will stop. The job will update to SCHEDULED until the startTime of the next maintenance window.

- **COMPLETED**

  A continuous job is handled in one of the following ways:

  - For a continuous job *without* the optional scheduling configuration selected, it's always in progress and continues to run for any new devices that are added to the target group. It will never reach a status state of COMPLETED.
  - For a continuous job *with* the optional scheduling configuration selected, the following is true:
    - If an endTime *was* provided, a continuous job will reach COMPLETED status when endTime has passed and all job executions have reached a terminal status state.
    - If an endTime *was not* provided in the optional scheduling configuration, the continuous job will continue to perform the job document rollout.

  For a snapshot job, the job status changes to COMPLETED when all of its job executions enter a terminal state, such as SUCCEEDED, FAILED, TIMED_OUT, REMOVED, or CANCELED.

- **CANCELED**

  When you cancel a job using the AWS IoT console, the CancelJob API, or the Job abort configuration, the job status changes to CANCELED. During job cancellation, AWS IoT Jobs starts canceling previously created job executions.

  For information about concurrency and limits that apply to jobs that are being canceled, see AWS IoT Jobs limits.

- **DELETION_IN_PROGRESS**

  When you delete a job using the AWS IoT console or the DeleteJob API, the job status changes to DELETION_IN_PROGRESS. During job deletion, AWS IoT Jobs starts deleting previously created job executions. After all job executions have been deleted, the job disappears from your AWS account.

Job execution states

The following table shows the different states of an AWS IoT job execution and whether the state change is initiated by the device or by AWS IoT Jobs.

Job execution states and source

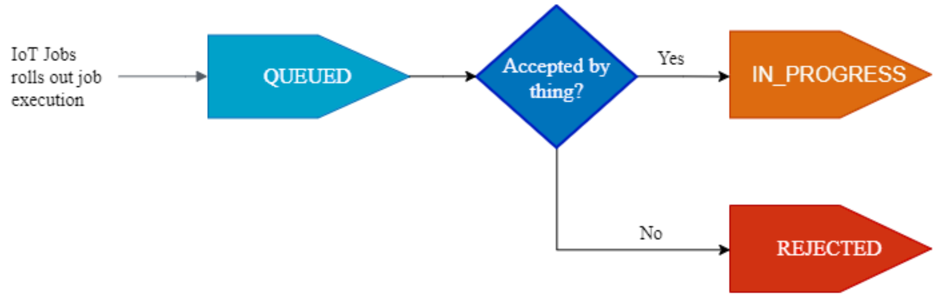
Job execution state	Initiated by device?	Initiated by AWS IoT Jobs?	Terminal status?	Can be retried?
`QUEUED`	No	Yes	No	Not applicable
`IN_PROGRESS`	Yes	No	No	Not applicable
`SUCCEEDED`	Yes	No	Yes	Not applicable
`FAILED`	Yes	No	Yes	Yes
`TIMED_OUT`	No	Yes	Yes	Yes
`REJECTED`	Yes	No	Yes	No
`REMOVED`	No	Yes	Yes	No
`CANCELED`	No	Yes	Yes	No

The following section describes more about the states of a job execution that's rolled out when you create a job with AWS IoT Jobs.

- **QUEUED**

  When AWS IoT Jobs rolls out a job execution for a target device, the job execution status is set to `QUEUED`. The job execution remains in the `QUEUED` state until:

  - Your device receives the job execution and invokes the Jobs API operations and reports the status as `IN_PROGRESS`.
  - You cancel the job or job execution, or when the abort criteria that you specified is met, and the status changes to `CANCELED`.
  - Your device is removed from the target group and the status changes to `REMOVED`.



- **IN_PROGRESS**

  If your IoT device subscribes to the reserved Job topics `$notify` and `$notify-next`, and your device invokes either the `StartNextPendingJobExecution` API or the `UpdateJobExecution` API with a status of `IN_PROGRESS`, AWS IoT Jobs will set the job execution status to `IN_PROGRESS`.

  The `UpdateJobExecution` API can be invoked multiple times with a status of `IN_PROGRESS`. You can specify additional details about the execution steps using the `statusDetails` object.

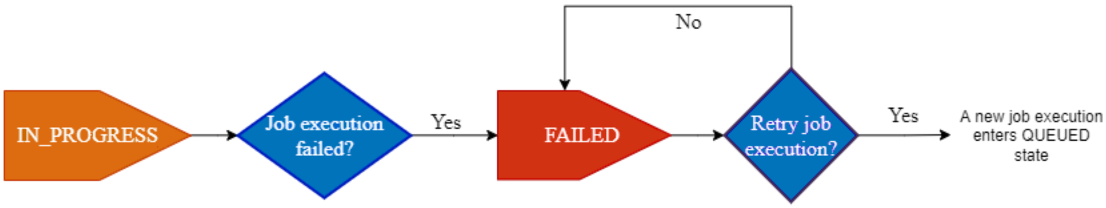
  > **Note**
  >
  > If you create multiple jobs for each device, AWS IoT Jobs and the MQTT protocol don't guarantee order of delivery.

- **SUCCEEDED**

  When your device successfully completes the remote operation, the device must invoke the `UpdateJobExecution` API with a status of `SUCCEEDED` to indicate that the job execution succeeded. AWS IoT Jobs then updates and returns the job execution status as `SUCCEEDED`.

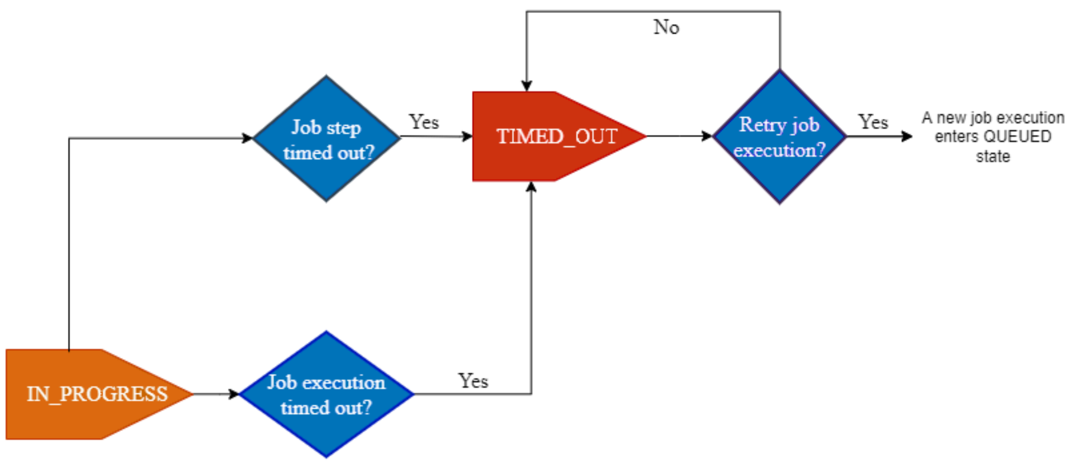
- **FAILED**

  When your device fails to complete the remote operation, the device must invoke the `UpdateJobExecution` API with a status of `Failed` to indicate that the job execution failed. AWS IoT Jobs then updates and returns the job execution status as `Failed`. You can retry this job execution for the device using the Job execution retry configuration.



- **TIMED_OUT**

  When your device fails to complete a job step when the status is `IN_PROGRESS`, or when it fails to complete the remote operation within the timeout duration of the in-progress timer, AWS IoT Jobs sets the job execution status to `TIMED_OUT`. You also have a step timer for each job step of an in-progress job and applies only to the job execution. The in-progress timer duration is specified using the `inProgressTimeoutInMinutes` property of the [Job execution timeout configuration](). You can retry this job execution for the device using the [Job execution retry configuration]().



- **REJECTED**

  When your device receives an invalid or incompatible request, the device must invoke the `UpdateJobExecution` API with a status of `REJECTED`. AWS IoT Jobs then updates and returns the job execution status as `REJECTED`.

- **REMOVED**

  When your device is no longer a valid target for the job execution, such as when it's detached from a dynamic thing group, AWS IoT Jobs sets the job execution status to `REMOVED`. You can re-attach the thing to your target group and restart the job execution for the device.

- **CANCELED**

When you cancel a job or cancel a job execution using the console or the `CancelJob` or `CancelJobExecution` API, or when the abort criteria specified using the Job abort configuration is met, AWS IoT Jobs cancels the job and sets the job execution status to `CANCELED`.

Managing jobs

Use jobs to notify devices of a software or firmware update. You can use the AWS IoT console, the Job management and control API operations, the AWS Command Line Interface, or the AWS SDKs to create and manage jobs.

Code signing for jobs

When sending code to devices, for devices to detect whether the code has been modified in transit, we recommend that you sign the code file by using the AWS CLI. For instructions, see Create and manage jobs by using the AWS CLI.

For more information, see What Is Code Signing for AWS IoT?.

Job document

Before you create a job, you must create a job document. If you're using code signing for AWS IoT, you must upload your job document to a versioned Amazon S3 bucket. For more information about creating an Amazon S3 bucket and uploading files to it, see Getting Started with Amazon Simple Storage Service in the *Amazon S3 Getting Started Guide*.

> **Tip**
>
> For job document examples, see the jobs-agent.js example in the AWS IoT SDK for JavaScript.

Presigned URLs

Your job document can contain a presigned Amazon S3 URL that points to your code file (or other file). Presigned Amazon S3 URLs are valid only for a limited amount of time and are generated when a device requests a job document. Because the presigned URL isn't created when you're creating the job document, use a placeholder URL in your job document instead. A placeholder URL looks like the following:

```
${aws:iot:s3-presigned-url-v2:https://
s3.region.amazonaws.com/<bucket>/<code file>}
```

where:

- *bucket* is the Amazon S3 bucket that contains the code file.
- *code file* is the Amazon S3 key of the code file.

When a device requests the job document, AWS IoT generates the presigned URL and replaces the placeholder URL with the presigned URL. Your job document is then sent to the device.

IAM role to grant permission to download files from S3

When you create a job that uses presigned Amazon S3 URLs, you must provide an IAM role. The role must grant permission to download files from the Amazon S3 bucket where the data or updates are stored. The role must also grant permission for AWS IoT to assume the role.

You can specify an optional timeout for the presigned URL. For more information, see CreateJob.

Grant AWS IoT Jobs permission to assume your role

1. Go to the Roles hub of the IAM console and choose your role.
2. On the **Trust Relationships** tab, choose **Edit Trust Relationship** and replace the policy document with the following JSON. Choose **Update Trust Policy**.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "iot.amazonaws.com"
        ]
      },
```

```
        "Action": "sts:AssumeRole"
      }
    ]
  }
```

3. To protect against the confused deputy problem, add the global condition context keys `aws:SourceArn` and `aws:SourceAccount` to the policy.

> **Important**
>
> Your `aws:SourceArn` must comply with the format: `arn:aws:iot:`*`region`*`:`*`account-id`*`:*`. Make sure that *`region`* matches your AWS IoT Region and *`account-id`* matches your customer account ID. For more information, see Cross-service confused deputy prevention.

```
{
  "Effect": "Allow",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service":
          "iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole",
      "Condition": {
         "StringEquals": {
            "aws:SourceAccount": "123456789012"
         },
         "ArnLike": {
              "aws:SourceArn": "arn:aws:iot:*:123456789012:job/*"
         }
       }
     }
   ]
}
```

4. If your job uses a job document that's an Amazon S3 object, choose **Permissions** and use the following JSON. This adds a policy that grants permission to download files from your Amazon S3 bucket:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::your_S3_bucket/*"
        }
    ]
}
```

Presigned URL for file upload

If your devices need to upload files to an Amazon S3 bucket during a job deployment, then you can include the following presigned URL placeholder in your job document:

```
${aws:iot:s3-presigned-url-upload:https://s3.region.amazonaws.com/<bucket>/<key>}
```

You can use a max of two of each of `${thingName}`, `${jobId}`, and `${executionNumber}` as reserved keywords within the `key` attribute in the file upload placeholder URL located in your job document. The local placeholder representing those reserved keywords in the `key` attribute will be parsed and replaced when the job execution is created. Using a local placeholder with reserved keywords specific to each device ensures each uploaded file from a device is specific to that device and not overwritten by a similar uploaded file from another device targeted by the same job deployment. For information on troubleshooting local placeholders within a presigned URL placeholder for uploading files during a job deployment, see General Troubleshooting Error Messages.

> **Note**
>
> The Amazon S3 bucket name can't contain the local placeholder representing the reserved keywords for the uploaded file. The local placeholder must be located in the `key` attribute.

This presigned URL placeholder will be converted to an Amazon S3 presigned upload URL in your job document when a device receives it. Your devices will use this to upload files to a destination Amazon S3 bucket.

> **Note**
>
> When the Amazon S3 bucket and key are not provided in the above placeholder URL, AWS IoT Jobs will automatically generate a key for each device using a max of two of each of `${thingName}`, `${jobId}`, and `${executionNumber}`.

Presigned URL using Amazon S3 versioning

Safeguarding the integrity of a file stored in an Amazon S3 bucket is critical for ensuring secure job deployments using that file to your device fleet. With the use of Amazon S3 versioning, you can add a version identifier for each variant of the file stored in your Amazon S3 bucket for tracking each verison of the file. This provides insight into what version of the file is deployed to your device fleet using AWS IoT Jobs. For more information on Amazon S3 buckets using versioning, see [Using versioning in Amazon S3 buckets](#).

If the file is stored in Amazon S3 and the job document contains a presigned URL placeholder, AWS IoT Jobs will generate a presigned URL in the job document using the Amazon S3 bucket, bucket key, and version of the file stored in the Amazon S3 bucket. This presigned URL generated in the job document will replace the presigned URL placeholder originally in the job document. If you update the file stored in your Amazon S3 bucket, a new version of the file and subsequent `versionId` will be created to signal the updates made and provide the ability to target that specific file in future job deployments.

Refer to the following examples for a before and during look of the Amazon S3 presigned URLs in your job document using the `versionId`:

Amazon S3 Presigned URL placeholder (Before Job Deployment)

```
//Virtual-hosted style URL
${aws:iot:s3-presigned-url-v2:https://bucket-name.s3.region-code.amazonaws.com/key-
name%3FversionId%3Dversion-id}

//Path-style URL
${aws:iot:s3-presigned-url-v2:https://s3.region-code.amazonaws.com/bucket-name/key-
name%3FversionId%3Dversion-id}
```

Amazon S3 Presigned URL (During Job Deployment)

```
//Virtual-hosted style URL
```

```
${aws:iot:s3-presigned-url-v2:https://sample-bucket-name.s3.us-west-2.amazonaws.com/sample-code-file.png%3FversionId%3Dversion1}

//Path-style
${aws:iot:s3-presigned-url-v2:https://s3.us-west-2.amazonaws.com/sample-bucket-name/sample-code-file.png%3FversionId%3Dversion1}
```

For more information on Amazon S3 virtual-hosted and path-style object URLs, see Virtual-hosted-style requests and Path-style requests.

Note

If you want to append `versionId` to a Amazon S3 presigned URL, it must conform to URL encoding supporting AWS SDK for Java 2.x. For more information, see Changes in parsing Amazon S3 URIs from version 1 to version 2.

Amazon S3 Presigned URL placeholder version differences

The following list outlines the differences between Amazon S3 presigned URL placeholders `${aws:iot:s3-presigned-url-v1` (version 1) and `${aws:iot:s3-presigned-url-v2` (version 2):

- The Amazon S3 presigned URL placeholder `${aws:iot:s3-presigned-url-v1` does not support `version-id`.
- The Amazon S3 presigned URL placeholder `${aws:iot:s3-presigned-url-v1` receives the Amazon S3 URL as unencoded. The Amazon S3 Presigned URL placeholder `${aws:iot:s3-presigned-url-v2` requires the Amazon S3 URL to be encoded to conform with the Amazon S3 SDK standard.

Topics

Create and manage jobs by using the AWS Management Console

This section describes how you can create and manage jobs from the AWS IoT console. After you've created a job, you can view information about the job on the details page, and manage the job.

> **Note**
>
> If you want to perform code signing for AWS IoT jobs, use the AWS CLI. For more information, see Create and manage jobs by using the AWS CLI.

Topics

Create manage jobs by using the AWS Management Console

To create a job, log in to the AWS IoT console, and go to the Jobs hub in the **Remote Actions** section. Then, perform the following steps.

1. On the **Jobs** page, in the **Jobs** dialog box, choose **Create job**.
2. Depending on the device that you're using, you can create a custom job, a FreeRTOS OTA update job, or an AWS IoT Greengrass job. For this example, choose **Create a custom job**. Choose **Next**.
3. On the **Custom job properties** page, in the **Job properties** dialog box, enter your information for the following fields:

   - **Name**: Enter a unique, alphanumeric job name.
   - **Description - optional**: Enter an optional description about your Job.
   - **Tags - optional**:

   > **Note**
   >
   > We recommend that you don't use personally identifiable information in your job IDs and description.

   Choose **Next**.
4. On the **File configuration** page in the **Job targets** dialog box, select the **Things** or **Thing groups** that you want to run this job.

In the **Job document** dialog box, select one of the following options:

- **From file**: A JSON job file you previously uploaded to an Amazon S3 bucket
  - **Code signing**

    In the job document located in your Amazon S3 URL, `${aws:iot:code-sign-signature:s3://region.bucket/code-file@code-file-version-id}` is required as a placeholder until it is replaced with the signed code file path using your **Code signing profile**. The new signed code file will initially appear in a `SignedImages` folder in your Amazon S3 source bucket. A new job document containing a `Codesigned_` prefix will be created with the signed code file path replacing the code-sign placeholder and placed in your Amazon S3 URL for creating a new job.
  - **Pre-sign resource URLs**

    In the **Pre-signing role** drop down, choose the IAM role you created in Presigned URLs. Using `${aws:iot:s3-presigned-url:` to presign URLs for objects located in Amazon S3 is a best security practice for devices downloading objects from Amazon S3.

    If you want to use presigned URLs for a code signing placeholder, use the following example template:

    ```
    ${aws:iot:s3-presigned-url:${aws:iot:code-sign-signature:<S3 URL>}
    ```
- **From template**: A job template containing a job document and job configurations. The job template can be a custom job template you created or an AWS managed template.

  If you're creating a job for performing frequently used remote actions such as rebooting your device, you can use an AWS managed template. These templates have already been preconfigured for use. For more information, see Create a custom job template and Create custom job templates from managed templates.

5. On the **Job configuration** page in the **Job configuration** dialog box, select one of the following job types:

   - **Snapshot job**: A snapshot job is complete when it's finished its run on the target devices and groups.
   - **Continuous job**: A continuous job applies to thing groups and runs on any device that you later add to a specified target group.

6. In the **Additional configurations - optional** dialog box, review the following optional Job configurations and make your selections accordingly:

   - **Rollout configuration**
   - **Scheduling configuration**
   - **Job executions timeout configuration**
   - **Job executions retry configuration - new**
   - **Abort configuration**

   Refer to the following sections for additional information on Job configurations:

   - Job rollout, scheduling, and abort configurations
   - Job execution timeout and retry configurations

   Review all of your job selections and then choose **Submit** to create your job.

View and manage jobs by using the AWS Management Console

After you create the job, the console generates a JSON signature and places it in your job document. You can use the AWS IoT console to view the status, cancel, or delete a job.

If you choose the job that you created, you can find:

- General job details, such as the job name, description, type, the time when it was created, last updated, and the estimated start time.
- Any job configurations that you specified and their status.
- The job document.
- The job executions and any optional tags that you specified.

To manage jobs, go to the Job hub of the console and choose whether you want to edit, delete, or cancel the job.

Create and manage jobs by using the AWS CLI

This section describes how to create and manage jobs.

Create jobs

To create an AWS IoT job, use the **CreateJob** command. The job is queued for execution on the targets (things or thing groups) that you specify. To create an AWS IoT job, you need a job document that can be included in the body of the request or as a link to an Amazon S3 document. If the job includes downloading files using presigned Amazon S3 URLs, you need an IAM role Amazon Resource Name (ARN) that has permission to download the file and grants permission to the AWS IoT Jobs service to assume the role.

For more information on the syntax when entering the date and time using an API command or the AWS CLI, see Timestamp.

Code signing with jobs

If you're using code signing for AWS IoT, you must start a code signing job and include the output in your job document. This will replace the code sign signature placeholder in your job document, which is required as a placeholder until it is replaced with the signed code file path using your **Code signing profile**. The code sign signature placeholder will look like the following:

```
    ${aws:iot:code-sign-signature:s3://region.bucket/code-file@code-file-version-id}
```

Use the start-signing-job command to create a code signing job. `start-signing-job` returns a job ID. To get the Amazon S3 location where the signature is stored, use the **describe-signing-job** command. You can then download the signature from Amazon S3. For more information about code signing jobs, see Code signing for AWS IoT.

Your job document must contain a presigned URL placeholder for your code file and the JSON signature output placed in an Amazon S3 bucket using the **start-signing-job** command:

```
{
    "presign": "${aws:iot:s3-presigned-url:https://s3.region.amazonaws.com/bucket/
image}",
}
```

Create a job with a job document

The following command shows how to create a job using a job document (*`job-document.json`*) stored in an Amazon S3 bucket (*`jobBucket`*), and a role with permission to download files from Amazon S3 (*`S3DownloadRole`*).

```
aws iot create-job  \
      --job-id 010  \
      --targets arn:aws:iot:us-east-1:123456789012:thing/thingOne  \
      --document-source https://s3.amazonaws.com/amzn-s3-demo-bucket/job-document.json
 \
      --timeout-config inProgressTimeoutInMinutes=100 \
      --job-executions-rollout-config "{ \"exponentialRate\": { \"baseRatePerMinute
\": 50, \"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings
\": 1000, \"numberOfSucceededThings\": 1000}}, \"maximumPerMinute\": 1000}" \
      --abort-config "{ \"criteriaList\": [ { \"action\": \"CANCEL\", \"failureType
\": \"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20},
 { \"action\": \"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings
\": 200, \"thresholdPercentage\": 50}]}" \
      --presigned-url-config "{\"roleArn\":\"arn:aws:iam::123456789012:role/
S3DownloadRole\", \"expiresInSec\":3600}"
```

The job is run on *thingOne*.

The optional `timeout-config` parameter specifies the amount of time each device has to finish its execution of the job. The timer starts when the job execution status is set to `IN_PROGRESS`. If the job execution status isn't set to another terminal state before the time expires, it's set to `TIMED_OUT`.

The in-progress timer can't be updated and applies to all job executions for the job. Whenever a job execution remains in the `IN_PROGRESS` state for longer than this interval, it fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

For more information about creating configurations for job rollouts and aborts, see Job Rollout and Abort Configuration.

Note

Job documents that are specified as Amazon S3 files are retrieved at the time you create the job. If you change the contents of the Amazon S3 file that you used as the source of your job document after you've created the job document, then what's sent to the job targets doesn't change.

Update a job

To update a job, use the **UpdateJob** command. You can update the `description`, `presignedUrlConfig`, `jobExecutionsRolloutConfig`, `abortConfig`, and `timeoutConfig` fields of a job.

```
aws iot update-job  \
  --job-id 010  \
  --description "updated description" \
  --timeout-config inProgressTimeoutInMinutes=100 \
  --job-executions-rollout-config "{ \"exponentialRate\": { \"baseRatePerMinute\": 50,
 \"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings\": 1000,
 \"numberOfSucceededThings\": 1000}, \"maximumPerMinute\": 1000}}" \
  --abort-config "{ \"criteriaList\": [ { \"action\": \"CANCEL\", \"failureType
\": \"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20},
 { \"action\": \"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings
\": 200, \"thresholdPercentage\": 50}]}" \
  --presigned-url-config "{\"roleArn\":\"arn:aws:iam::123456789012:role/
S3DownloadRole\", \"expiresInSec\":3600}"
```

For more information, see [Job Rollout and Abort Configuration](#).

Cancel a job

To cancel a job, use the **CancelJob** command. Canceling a job stops AWS IoT from rolling out any new job executions for the job. It also cancels any job executions that are in a QUEUED state. AWS IoT keeps any job executions in a terminal state untouched because the device has already completed the job. If the status of a job execution is `IN_PROGRESS`, it also remains untouched unless you use the optional `--force` parameter.

The following command shows how to cancel a job with ID 010.

```
aws iot cancel-job --job-id 010
```

The command displays the following output:

```
{
    "jobArn": "string",
    "jobId": "string",
    "description": "string"
```

```
}
```

When you cancel a job, job executions that are in a QUEUED state are canceled. Job executions that are in an IN_PROGRESS state are canceled, but only if you specify the optional --force parameter. Job executions in a terminal state aren't canceled.

> **Warning**
>
> Canceling a job that's in the IN_PROGRESS state (by setting the --force parameter) cancels any job executions that are in progress and causes the device that's running the job to be unable to update the job execution status. Use caution and make sure that each device executing a canceled job can recover to a valid state.

The status of a canceled job or of one of its job executions is eventually consistent. AWS IoT stops scheduling new job executions and QUEUED job executions for that job to devices as soon as possible. Changing the status of a job execution to CANCELED might take some time, depending on the number of devices and other factors.

If a job is canceled because it's met the criteria defined by an AbortConfig object, the service adds auto-populated values for the comment and reasonCode fields. You can create your own values for reasonCode when the job cancellation is user-driven.

Cancel a job execution

To cancel a job execution on a device, use the **CancelJobExecution** command. It cancels a job execution that's in a QUEUED state. If you want to cancel a job execution that's in progress, you must use the --force parameter.

The following command shows how to cancel the job execution from job 010 running on myThing.

```
aws iot cancel-job-execution --job-id 010 --thing-name myThing
```

The command displays no output.

A job execution that's in a QUEUED state is canceled. A job execution that's in an IN_PROGRESS state is canceled, but only if you specify the optional --force parameter. Job executions in a terminal state can't be canceled.

> **Warning**
>
> When you cancel a job execution that's in the `IN_PROGRESS` state, the device can't update the job execution status. Use caution and make sure that the device can recover to a valid state.

If the job execution is in a terminal state, or if the job execution is in an `IN_PROGRESS` state and the `--force` parameter isn't set to `true`, this command causes an `InvalidStateTransitionException`.

The status of a canceled job execution is eventually consistent. Changing the status of a job execution to `CANCELED` might take some time, depending on various factors.

Delete a job

To delete a job and its job executions, use the **DeleteJob** command. By default, you can only delete a job that's in a terminal state (`SUCCEEDED` or `CANCELED`). Otherwise, an exception occurs. You can delete a job in the `IN_PROGRESS` state, however, only if the `force` parameter is set to `true`.

To delete a job, run the following command:

```
aws iot delete-job --job-id 010 --force|--no-force
```

The command displays no output.

> **Warning**
>
> When you delete a job that's in the `IN_PROGRESS` state, the device that's deploying the job can't access job information or update the job execution status. Use caution and make sure that each device deploying a job that's been deleted can recover to a valid state.

It can take some time to delete a job, depending on the number of job executions created for the job and other factors. While the job is being deleted, `DELETION_IN_PROGRESS` appears as the status of the job. An error results if you attempt to delete or cancel a job with a status that's already `DELETION_IN_PROGRESS`.

Only 10 jobs can have a status of `DELETION_IN_PROGRESS` at the same time. Otherwise, a `LimitExceededException` occurs.

Get a job document

To retrieve a job document for a job, use the **GetJobDocument** command. A job document is a description of the remote operations to be performed by the devices.

To get a job document, run the following command:

```
aws iot get-job-document --job-id 010
```

The command returns the job document for the specified job:

```
{
    "document": "{\n\t\"operation\":\"install\",\n\t\"url\":\"http://amazon.com/
firmWareUpate-01\",\n\t\"data\":\"${aws:iot:s3-presigned-url:https://s3.amazonaws.com/
amzn-s3-demo-bucket/datafile}\"\n}"
}
```

> **Note**
>
> When you use this command to retrieve a job document, placeholder URLs aren't replaced by presigned Amazon S3 URLs. When a device calls the GetPendingJobExecutions API operation, the placeholder URLs are replaced by presigned Amazon S3 URLs in the job document.

List jobs

To get a list of all jobs in your AWS account, use the **ListJobs** command. Job data and job execution data are retained for a limited time. Run the following command to list all jobs in your AWS account:

```
aws iot list-jobs
```

The command returns all jobs in your account, sorted by the job status:

```
{
    "jobs": [
```

```
        {
            "status": "IN_PROGRESS",
            "lastUpdatedAt": 1486687079.743,
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/013",
            "createdAt": 1486687079.743,
            "targetSelection": "SNAPSHOT",
            "jobId": "013"
        },
        {
            "status": "SUCCEEDED",
            "lastUpdatedAt": 1486685868.444,
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/012",
            "createdAt": 1486685868.444,
            "completedAt": 148668789.690,
            "targetSelection": "SNAPSHOT",
            "jobId": "012"
        },
        {
            "status": "CANCELED",
            "lastUpdatedAt": 1486678850.575,
            "jobArn": "arn:aws:iot:us-east-1:123456789012:job/011",
            "createdAt": 1486678850.575,
            "targetSelection": "SNAPSHOT",
            "jobId": "011"
        }
    ]
}
```

Describe a job

To get the status of a job, run the **DescribeJob** command. The following command shows how to describe a job:

```
$ aws iot describe-job --job-id 010
```

The command returns the status of the specified job. For example:

```
{
    "documentSource": "https://s3.amazonaws.com/amzn-s3-demo-bucket/job-
document.json",
    "job": {
        "status": "IN_PROGRESS",
        "jobArn": "arn:aws:iot:us-east-1:123456789012:job/010",
```

```
        "targets": [
            "arn:aws:iot:us-east-1:123456789012:thing/myThing"
        ],
        "jobProcessDetails": {
            "numberOfCanceledThings": 0,
            "numberOfFailedThings": 0,
            "numberOfInProgressThings": 0,
            "numberOfQueuedThings": 0,
            "numberOfRejectedThings": 0,
            "numberOfRemovedThings": 0,
            "numberOfSucceededThings": 0,
            "numberOfTimedOutThings": 0,
            "processingTargets": [
                arn:aws:iot:us-east-1:123456789012:thing/thingOne,
                arn:aws:iot:us-east-1:123456789012:thinggroup/thinggroupOne,
                arn:aws:iot:us-east-1:123456789012:thing/thingTwo,
                arn:aws:iot:us-east-1:123456789012:thinggroup/thinggroupTwo
            ]
        },
        "presignedUrlConfig": {
            "expiresInSec": 60,
            "roleArn": "arn:aws:iam::123456789012:role/S3DownloadRole"
        },
        "jobId": "010",
        "lastUpdatedAt": 1486593195.006,
        "createdAt": 1486593195.006,
        "targetSelection": "SNAPSHOT",
        "jobExecutionsRolloutConfig": {
            "exponentialRate": {
                "baseRatePerMinute": integer,
                "incrementFactor": integer,
                "rateIncreaseCriteria": {
                    "numberOfNotifiedThings": integer, // Set one or the other
                    "numberOfSucceededThings": integer // of these two values.
                },
            "maximumPerMinute": integer
         }
        },
        "abortConfig": {
            "criteriaList": [
                {
                    "action": "string",
                    "failureType": "string",
                    "minNumberOfExecutedThings": integer,
```

```
                    "thresholdPercentage": integer
                }
            ]
        },
        "timeoutConfig": {
           "inProgressTimeoutInMinutes": number
          }
    }
}
```

List executions for a job

A job running on a specific device is represented by a job execution object. Run the **ListJobExecutionsForJob** command to list all job executions for a job. The following shows how to list the executions for a job:

```
aws iot list-job-executions-for-job --job-id 010
```

The command returns a list of job executions:

```
{
    "executionSummaries": [
    {
        "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingOne",
        "jobExecutionSummary": {
            "status": "QUEUED",
            "lastUpdatedAt": 1486593196.378,
            "queuedAt": 1486593196.378,
            "executionNumber": 1234567890
        }
    },
    {
        "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingTwo",
        "jobExecutionSummary": {
            "status": "IN_PROGRESS",
            "lastUpdatedAt": 1486593345.659,
            "queuedAt": 1486593196.378,
            "startedAt": 1486593345.659,
            "executionNumber": 4567890123
        }
    }
    ]
```

```
}
```

List job executions for a thing

Run the **ListJobExecutionsForThing** command to list all job executions running on a thing. The following shows how to list job executions for a thing:

```
aws iot list-job-executions-for-thing --thing-name thingOne
```

The command returns a list of job executions that are running or have run on the specified thing:


```
{
    "executionSummaries": [
    {
        "jobExecutionSummary": {
            "status": "QUEUED",
            "lastUpdatedAt": 1486687082.071,
            "queuedAt": 1486687082.071,
            "executionNumber": 9876543210
        },
        "jobId": "013"
    },
    {
        "jobExecutionSummary": {
            "status": "IN_PROGRESS",
            "startAt": 1486685870.729,
            "lastUpdatedAt": 1486685870.729,
            "queuedAt": 1486685870.729,
            "executionNumber": 1357924680
        },
        "jobId": "012"
    },
    {
        "jobExecutionSummary": {
            "status": "SUCCEEDED",
            "startAt": 1486678853.415,
            "lastUpdatedAt": 1486678853.415,
            "queuedAt": 1486678853.415,
            "executionNumber": 4357680912
        },
        "jobId": "011"
    },
```

```
    {
        "jobExecutionSummary": {
            "status": "CANCELED",
            "startAt": 1486593196.378,
            "lastUpdatedAt": 1486593196.378,
            "queuedAt": 1486593196.378,
            "executionNumber": 2143174250
        },
        "jobId": "010"
    }
    ]
}
```

Describe job execution

Run the **DescribeJobExecution** command to get the status of a job execution. You must specify a job ID and thing name and, optionally, an execution number to identify the job execution. The following shows how to describe a job execution:

```
aws iot describe-job-execution --job-id 017 --thing-name thingOne
```

The command returns the `JobExecution`. For example:

```
{
    "execution": {
        "jobId": "017",
        "executionNumber": 4516820379,
        "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/thingOne",
        "versionNumber": 123,
        "createdAt": 1489084805.285,
        "lastUpdatedAt": 1489086279.937,
        "startedAt": 1489086279.937,
        "status": "IN_PROGRESS",
        "approximateSecondsBeforeTimedOut": 100,
        "statusDetails": {
            "status": "IN_PROGRESS",
            "detailsMap": {
                "percentComplete": "10"
            }
        }
    }
}
```

Delete job execution

Run the **DeleteJobExecution** command to delete a job execution. You must specify a job ID, a thing name, and an execution number to identify the job execution. The following shows how to delete a job execution:

```
aws iot delete-job-execution --job-id 017 --thing-name thingOne --execution-number
 1234567890 --force|--no-force
```

The command displays no output.

By default, the status of the job execution must be `QUEUED` or in a terminal state (`SUCCEEDED`, `FAILED`, `REJECTED`, `TIMED_OUT`, `REMOVED`, or `CANCELED`). Otherwise, an error occurs. To delete a job execution with a status of `IN_PROGRESS`, you can set the `force` parameter to `true`.

> **Warning**
>
> When you delete a job execution with a status of `IN_PROGRESS`, the device that's executing the job can't access job information or update the job execution status. Use caution and make sure that the device can recover to a valid state.

Job templates

Use job templates to preconfigure jobs that you can deploy to multiple sets of target devices. To deploy frequently performed remote actions to your devices, like rebooting or installing an application, you can use templates to define standard configurations. To perform operations such as deploying security patches and bug fixes, you can create templates from existing jobs.

When creating a job template, specify the following additional configurations and resources.

- Job properties
- Job documents and targets
- Rollout, scheduling, and cancel criteria
- Timeout and retry criteria

Custom and AWS managed templates

Depending on the remote action that you want to perform, you can either create a custom job template or use an AWS managed template. Use custom job templates to provide your own custom job document and create reusable jobs to deploy to your devices. AWS managed templates are job templates provided by AWS IoT Jobs for commonly performed actions. These templates have a predefined job document for some remote actions so you don't have to create your own job document. Managed templates help you create reusable jobs for faster launch to your devices.

Topics

Use AWS managed templates to deploy common remote operations

AWS managed templates are job templates provided by AWS. They're used for frequently performed remote actions such as rebooting, downloading a file, or installing an application on your devices. These templates have a predefined job document for each remote action so you don't have to create your own job document.

You can choose from a set of predefined configurations and create jobs using these templates without writing any additional code. Using managed templates, you can view the job document deployed to your fleets. You can create a job using these templates and create a custom job template that you can reuse for your remote operations.

What do managed templates contain?

Each AWS managed template contains:

- The environment to run the commands in the job document.
- A job document that specifies the name of the operation and its parameters. For example, if you use a **Download file** template, the operation name is *Download file* and the parameters can be:
  - The URL of the file that you want to download to your device. This can be an internet resource or a public or pre-signed Amazon Simple Storage Service((Amazon S3) URL.
  - A local file path on the device to store the downloaded file.

  For more information about the job documents and its parameters, see Managed template remote actions and job documents.

Prerequisites

For your devices to run the remote actions specified by the managed template job document, you must:

- **Install the specific software on your device**

  Use your own device software and job handlers, or the AWS IoT Device Client. Depending on your business case, you can also run them both so that they perform different functions.

  - **Use your own device software and job handlers**

    You can write your own code for the devices by using the AWS IoT Device SDK and its library of handlers that support the remote operations. To deploy and run jobs, verify that the device agent libraries have been installed correctly and are running on the devices.

    You can also choose to use your own handlers that support the remote operations. For more information, see Sample job handlers in the AWS IoT Device Client GitHub repository.

  - **Use the AWS IoT Device Client**

    Or, you can install and run the AWS IoT Device Client on your devices because it supports using all managed templates directly from the console by default.

    The Device Client is an open-source software written in C++ that you can compile and install on your embedded Linux-based IoT devices. The Device Client has a *base client* and discrete *client-side features*. The base client establishes connectivity with AWS IoT over MQTT protocol and can connect with the different client-side features.

    To perform remote operations on your devices, use the *client-side Jobs feature* of the Device Client. This feature contains a parser to receive the job document and job handlers that implement the remote actions specified in the job document. For more information about the Device Client and its features, see AWS IoT Device Client.

    When running on devices, the Device Client receives the job document and has a platform-specific implementation that it uses to run commands in the document. For more information about setting up the Device Client and using the Jobs feature, see AWS IoT tutorials.

- **Use a supported environment**

  For each managed template, you'll find information about the environment that you can use to run the remote actions. We recommend that you use the template with a supported Linux environment as specified in the template. Use the AWS IoT Device Client to run the managed template remote actions because it supports common microprocessors and Linux environments, like Debian and Ubuntu.

Managed template remote actions and job documents

The following section lists the different AWS managed templates for AWS IoT Jobs, and describes the remote actions that can be performed on the devices. The following section has information about the job document and a description of the job document parameters for each remote action. Your device-side software uses the template name and the parameters to perform the remote action.

AWS managed templates accept input parameters for which you specify a value when creating a job using the template. All managed templates have two optional input parameters in common: `runAsUser` and `pathToHandler`. Except for the `AWS-Reboot` template, the templates require additional input parameters for which you must specify a value when creating a job using the template. These required input parameters vary depending on the template that you choose. For example, if you choose the `AWS-Download-File` template, you must specify a list of packages to install, and a URL to download files from.

Specify a value for the input parameters when using the AWS IoT console or the AWS Command Line Interface (AWS CLI) to create a job that uses a managed template. When using the CLI, provide these values by using the `document-parameters` object. For more information, see documentParameters.

> **Note**
>
> Use `document-parameters` only when creating jobs from AWS managed templates. This parameter can't be used with custom job templates or to create jobs from them.

The following shows a description of the common optional input parameters. You'll see a description of other input parameters that each managed template requires in the next section.

`runAsUser`

This parameter specifies whether to run the job handler as another user. If it's not specified during job creation, the job handler is run as the same user as the Device Client. When you run the job handler as another user, specify a string value that's not longer than 256 characters.

`pathToHandler`

The path to the job handler running on the device. If it's not specified during job creation, the Device Client uses the current working directory.

The following shows the different remote actions, their job documents, and parameters that they accept. All these templates support the Linux environment for running the remote operation on the device.

AWS–Download–File

Template name

`AWS-Download-File`

Template description

A managed template provided by AWS for downloading a file.

Input parameters

This template has the following required parameters. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`downloadUrl`

The URL to download the file from. This can be an internet resource, an object in Amazon S3 that can be publicly accessed, or an object in Amazon S3 that can only be accessed by your device using a presigned URL. For more information about using presigned URLs and granting permissions, see Presigned URLs.

`filePath`

A local file path that shows the location in the device to store the downloaded file.

Device behavior

The device downloads the file from the specified location, verifies that the download is complete, and stores it locally.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `download-file.sh`, that the job handler must run to download the file. It also shows the required parameters `downloadUrl` and `filePath`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Download-File",
        "type": "runHandler",
        "input": {
          "handler": "download-file.sh",
          "args": [
            "${aws:iot:parameter:downloadUrl}",
            "${aws:iot:parameter:filePath}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Install–Application

Template name

`AWS-Install-Application`

Template description

A managed template provided by AWS for installing one or more applications.

Input parameters

This template has the following required parameter, `packages`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`packages`

A space-separated list of one or more applications to be installed.

Device behavior

The device installs the applications as specified in the job document.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `install-packages.sh`, that the job handler must run to download the file. It also shows the required parameter `packages`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Install-Application",
        "type": "runHandler",
        "input": {
          "handler": "install-packages.sh",
          "args": [
            "${aws:iot:parameter:packages}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Reboot

Template name

`AWS-Reboot`

Template description

A managed template provided by AWS for rebooting your device.

Input parameters

This template has no required parameters. You can specify the optional parameters `runAsUser` and `pathToHandler`.

Device behavior

The device reboots successfully.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `reboot.sh`, that the job handler must run to reboot the device.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Reboot",
        "type": "runHandler",
        "input": {
          "handler": "reboot.sh",
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Remove–Application

Template name

`AWS-Remove-Application`

Template description

A managed template provided by AWS for uninstalling one or more applications.

Input parameters

This template has the following required parameter, `packages`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`packages`

A space-separated list of one or more applications to be uninstalled.

Device behavior

The device uninstalls the applications as specified in the job document.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `remove-packages.sh`, that the job handler must run to download the file. It also shows the required parameter `packages`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Remove-Application",
        "type": "runHandler",
        "input": {
          "handler": "remove-packages.sh",
          "args": [
            "${aws:iot:parameter:packages}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Restart–Application

Template name

`AWS-Restart-Application`

Template description

A managed template provided by AWS for stopping and restarting one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

Services

A space-separated list of one or more applications to be restarted.

Device behavior

The specified applications are stopped and then restarted on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `restart-services.sh`, that the job handler must run to restart the system services. It also shows the required parameter `services`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Restart-Application",
        "type": "runHandler",
        "input": {
          "handler": "restart-services.sh",
          "args": [
            "${aws:iot:parameter:services}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Start–Application

Template name

`AWS-Start-Application`

Template description

A managed template provided by AWS for starting one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`services`

A space-separated list of one or more applications to be started.

Device behavior

The specified applications start running on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `start-services.sh`, that the job handler must run to start the system services. It also shows the required parameter `services`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Start-Application",
        "type": "runHandler",
        "input": {
          "handler": "start-services.sh",
          "args": [
            "${aws:iot:parameter:services}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Stop–Application

Template name

`AWS–Stop–Application`

Template description

A managed template provided by AWS for stopping one or more services.

Input parameters

This template has the following required parameter, `services`. You can also specify the optional parameters `runAsUser` and `pathToHandler`.

`services`

A space-separated list of one or more applications to be stopped.

Device behavior

The specified applications stop running on the device.

Job document

The following shows the job document and its latest version. The template shows the path to the job handler and the shell script, `stop-services.sh`, that the job handler must run to stop the system services. It also shows the required parameter `services`.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Stop-Application",
        "type": "runHandler",
        "input": {
          "handler": "stop-services.sh",
          "args": [
            "${aws:iot:parameter:services}"
          ],
          "path": "${aws:iot:parameter:pathToHandler}"
```

```
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

AWS–Run–Command

Template name

`AWS-Run-Command`

Template description

A managed template provided by AWS for running a shell command.

Input parameters

This template has the following required parameter, `command`. You can also specify the optional parameter `runAsUser`.

`command`

A comma separated string of command. Any comma contained in the command itself must be escaped.

Device behavior

The device runs the shell command as specified in the job document.

Job document

The following shows the job document and its latest version. The template shows the path to the job command and the command that you provided which the device will run.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
```

```
          "name": "Run-Command",
          "type": "runCommand",
          "input": {
            "command": "${aws:iot:parameter:command}"
          },
          "runAsUser": "${aws:iot:parameter:runAsUser}"
        }
      }
    ]
  }
```

Topics

- [Create a job from AWS managed templates by using the AWS Management Console](#)
- [Create a job from AWS managed templates by using the AWS CLI](#)

Create a job from AWS managed templates by using the AWS Management Console

Use the AWS Management Console to get information about AWS managed templates and create a job by using these templates. You can then save the job you create as your own custom template.

Get details about managed templates

You can get information about the different managed templates that are available to use from the AWS IoT console.

1. To see your available managed templates, go to the Job templates hub of the AWS IoT console and choose the **Managed templates** tab.
2. To view details, choose a managed template.

The details page contains the following information:

- Name, description, and Amazon Resource Name (ARN) of the managed template.
- The environment on which the remote operations can be performed, such as Linux.
- The JSON job document that specifies the path to the job handler and the commands to run on the device. For example, the following shows an example job document for the **AWS-Reboot** template. The template shows the path to the job handler and the shell script, `reboot.sh`, that the job handler must run to reboot the device.

```
{
  "version": "1.0",
  "steps": [
    {
      "action": {
        "name": "Reboot",
        "type": "runHandler",
        "input": {
          "handler": "reboot.sh",
          "path": "${aws:iot:parameter:pathToHandler}"
        },
        "runAsUser": "${aws:iot:parameter:runAsUser}"
      }
    }
  ]
}
```

For more information about the job document and its parameters for various remote actions, see Managed template remote actions and job documents.

- The latest version of the job document.

Create a job using managed templates

You can use the AWS Management console to choose an AWS managed template to use to create a job. This section shows you how.

You can also start the job creation workflow and then choose the AWS managed template that you want to use while creating the job. For more information about this workflow, see Create and manage jobs by using the AWS Management Console.

1. **Choose your AWS managed template**

   Go to the Job templates hub of the AWS IoT console, choose the **Managed templates** tab, and then choose your template.

2. **Create a job using your managed template**

   1. In the details page of your template, choose **Create job**.

      The console switches to the **Custom job properties** step of the **Create job** workflow where your template configuration has been added.

2. Enter a unique alphanumeric job name, and optional description and tags, and then choose **Next**.
3. Choose the things or thing groups as job targets that you want to run in this job.
4. In the **Job document** section, your template is displayed with its configuration settings and input parameters. Enter values for the input parameters of your chosen template. For example, if you chose the **AWS-Download-File** template:
   - For **downloadUrl**, enter the URL of the file to download, for example: `https://example.com/index.html`.
   - For **filePath**, enter the path on the device to store the downloaded file, for example: *`path/to/file`*.

   You can also optionally enter values for the parameters `runAsUser` and `pathToHandler`. For more information about the input parameters of each template, see Managed template remote actions and job documents.
5. On the **Job configuration** page, choose the job type as continuous or a snapshot job. A snapshot job is complete when it finishes its run on the target devices and groups. A continuous job applies to thing groups and runs on any device that you add to a specified target group.
6. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:
   - Job rollout, scheduling, and abort configurations
   - Job execution timeout and retry configurations

Create custom job templates from managed templates

You can use an AWS managed template and a custom job as a starting point to create your own custom job template. To create a custom job template, first create a job from your AWS managed template as described in the previous section.

You can then save the custom job as a template to create your own custom job template. To save as template:

1. Go to the Job hub of the AWS IoT console and choose the job containing your managed template.
2. Choose **Save as a job template** and then create your custom job template. For more information about creating a custom job template, see Create a job template from an existing job.

Create a job from AWS managed templates by using the AWS CLI

Use the AWS CLI to get information about AWS managed templates and create a job by using these templates. You can then save the job as a template and then create your own custom template.

List managed templates

The `list-managed-job-templates` AWS CLI command lists all of the job templates in your AWS account.

```
aws iot list-managed-job-templates
```

By default, running this command displays all available AWS managed templates and their details.

```
{
    "managedJobTemplates": [
        {
            "templateArn": "arn:aws:iot:region::jobtemplate/AWS-Reboot:1.0",
            "templateName": "AWS-Reboot",
            "description": "A managed job template for rebooting the device.",
            "environments": [
                "LINUX"
            ],
            "templateVersion": "1.0"
        },
        {
            "templateArn": "arn:aws:iot:region::jobtemplate/AWS-Remove-
Application:1.0",
            "templateName": "AWS-Remove-Application",
            "description": "A managed job template for uninstalling one or more
 applications.",
            "environments": [
                "LINUX"
            ],
            "templateVersion": "1.0"
        },
        {
            "templateArn": "arn:aws:iot:region::jobtemplate/AWS-Stop-Application:1.0",
            "templateName": "AWS-Stop-Application",
            "description": "A managed job template for stopping one or more system
 services.",
```

```
            "environments": [
                "LINUX"
            ],
            "templateVersion": "1.0"
        },

        ...

        {
            "templateArn": "arn:aws:iot:us-east-1::jobtemplate/AWS-Restart-
Application:1.0",
            "templateName": "AWS-Restart-Application",
            "description": "A managed job template for restarting one or more system
 services.",
            "environments": [
                "LINUX"
            ],
            "templateVersion": "1.0"
        }
    ]
}
```

For more information, see ListManagedJobTemplates.

Get details about a managed template

The `describe-managed-job-template` AWS CLI command gets details about a specified job template. Specify the job template name and an optional template version. If the template version is not specified, the predefined, default version is returned. The following shows an example of running the command to get details about the `AWS-Download-File` template.

```
aws iot describe-managed-job-template \
    --template-name AWS-Download-File
```

The command displays the template details and ARN, its job document, and the `documentParameters` parameter, which is a list of key-value pairs of input parameters of the template. For information about the different templates and input parameters, see Managed template remote actions and job documents.

Note

The `documentParameters` object returned when you use this API must only be used when creating jobs from AWS managed templates. The object must not be used for custom job templates. For an example that shows how to use this parameter, see Create a job by using managed templates.

```
{
    "templateName": "AWS-Download-File",
    "templateArn": "arn:aws:iot:region::jobtemplate/AWS-Download-File:1.0",
    "description": "A managed job template for downloading a file.",
    "templateVersion": "1.0",
    "environments": [
        "LINUX"
    ],
    "documentParameters": [
        {
            "key": "downloadUrl",
            "description": "URL of file to download.",
            "regex": "(.*?)",
            "example": "http://www.example.com/index.html",
            "optional": false
        },
        {
            "key": "filePath",
            "description": "Path on the device where downloaded file is written.",
            "regex": "(.*?)",
            "example": "/path/to/file",
            "optional": false
        },
        {
            "key": "runAsUser",
            "description": "Execute handler as another user. If not specified, then
 handler is executed as the same user as device client.",
            "regex": "(.){0,256}",
            "example": "user1",
            "optional": true
        },
        {
            "key": "pathToHandler",
```

```
            "description": "Path to handler on the device. If not specified, then
 device client will use the current working directory.",
            "regex": "(.){0,4096}",
            "example": "/path/to/handler/script",
            "optional": true
        }
    ],
    "document": "{\"version\":\"1.0\",\"steps\":[{\"action\":{\"name
\":\"Download-File\",\"type\":\"runHandler\",\"input\":{\"handler\":
\"download-file.sh\",\"args\":[\"${aws:iot:parameter:downloadUrl}\",
\"${aws:iot:parameter:filePath}\"],\"path\":\"${aws:iot:parameter:pathToHandler}\"},
\"runAsUser\":\"${aws:iot:parameter:runAsUser}\"}}]}"
}
```

For more information, see DescribeManagedJobTemplate.

Create a job by using managed templates

The `create-job` AWS CLI command can be used to create a job from a job template. It targets a device named `thingOne` and specifies the Amazon Resource Name (ARN) of the managed template to use as the basis for the job. You can override advanced configurations, such as timeout and cancel configurations, by passing the associated parameters of the `create-job` command.

The example shows how to create a job that uses the `AWS-Download-File` template. It also shows how to specify the input parameters of the template by using the `document-parameters` parameter.

> **Note**
>
> Use the `document-parameters` object only with AWS managed templates. This object must not be used with custom job templates.

```
aws iot create-job \
   --targets arn:aws:iot:region:account-id:thing/thingOne \
   --job-id "new-managed-template-job" \
   --job-template-arn arn:aws:iot:region::jobtemplate/AWS-Download-File:1.0 \
   --document-parameters downloadUrl=https://example.com/index.html,filePath=path/to/
file
```

where:

- *region* is the AWS Region.
- *account-id* is the unique AWS account number.
- *thingOne* is the name of the IoT thing for which the job is targeted.
- *AWS-Download-File:1.0* is the name of the managed template.
- `https://`*example.com/index*`.html` is the URL to download the file from.
- `https://`*pathto/file/index* is the path on the device to store the downloaded file.

Run the following command to create a job for the template, *AWS-Download-File*.

```
{
    "jobArn": "arn:aws:iot:region:account-id:job/new-managed-template-job",
    "jobId": "new-managed-template-job",
    "description": "A managed job template for downloading a file."
}
```

Create a custom job template from managed templates

1. Create a job using a managed template as described in the previous section.
2. Create a custom job template by using the ARN of the job that you created. For more information, see Create a job template from an existing job.

Create custom job templates

You can create job templates by using the AWS CLI and the AWS IoT console. You can also create jobs from job templates by using the AWS CLI, the AWS IoT console, and Fleet Hub for AWS IoT Device Management web applications. For more information about working with job templates in Fleet Hub applications, see Working with job templates in Fleet Hub for AWS IoT Device Management.

> **Note**
>
> The total number of substitution patterns in a job document should be less than or equal to ten.

Topics

Create custom job templates by using the AWS Management Console

This topic explains how to create, delete, and view details about job templates by using the AWS IoT console.

Create a custom job template

You can either create an original custom job template or create a job template from an existing job. You can also create a custom job template from an existing job that was created using an AWS managed template. For more information, see Create custom job templates from managed templates.

Create an original job template

1. **Start creating your job template**

   1. Go to the Job templates hub of the AWS IoT console and choose the **Custom templates** tab.
   2. Choose **Create job template**.

   > **Note**
   >
   > You can also navigate to the **Job templates** page from the **Related services** page under **Fleet Hub**.

2. **Specify job template properties**

   In the **Create job template** page, enter an alphanumeric identifier for your job name and an alphanumeric description to provide additional details about the template.

   > **Note**
   >
   > We don't recommend using personally identifiable information in your job IDs or descriptions.

3. **Provide job document**

   Provide a JSON job file that is either stored in an S3 bucket or as an inline job document that is specified within the job. This job file will become the job document when you create a job using this template.

   If the job file is stored in an S3 bucket, enter the S3 URL or choose **Browse S3**, and then navigate to your job document and select it.

   > **Note**
   >
   > You can select only S3 buckets in your current Region.

4. Continue to add any additional configurations for your job and then review and create your job. For information about the additional, optional configurations, refer to the following links:

   - Job rollout, scheduling, and abort configurations
   - Job execution timeout and retry configurations

Create a job template from an existing job

1. **Choose your job**

   1. Go to the Job hub of the AWS IoT console and choose the job that you want to use as the basis for your job template.
   2. Choose **Save as a job template**.

   > **Note**
   >
   > Optionally, you can choose a different job document or edit the advanced configurations from the original job, and then choose **Create job template**. Your new job template appears on the **Job templates** page.

2. **Specify job template properties**

   In the **Create job template** page, enter an alphanumeric identifier for your job name and an alphanumeric description to provide additional details about the template.

> **Note**
>
> The job document is the job file that you specified when creating the template. If the job document is specified within the job instead of an S3 location, you can see the job document in the details page of this job.

3. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:

   - Job rollout, scheduling, and abort configurations
   - Job execution timeout and retry configurations

Create a job from a custom job template

You can create a job from a custom job template by going to the details page of your job template as described in this topic. You can also create a job or by choosing the job template you want to use when running the job creation workflow. For more information, see Create and manage jobs by using the AWS Management Console.

This topic shows how to create a job from the details page of a custom job template. You can also create a job from an AWS managed template. For more information, see Create a job using managed templates.

1. **Choose your custom job template**

   Go to the Job templates hub of the AWS IoT console and choose the **Custom templates** tab, and then choose your template.

2. **Create a job using your custom template**

   To create a job:

   1. In the details page of your template, choose **Create job**.

      The console switches to the **Custom job properties** step of the **Create job** workflow where your template configuration has been added.

   2. Enter a unique alphanumeric job name, and optional description and tags, and then choose **Next**.

   3. Choose the things or thing groups as job targets that you want to run in this job.

In the **Job document** section, your template is displayed with its configuration settings. If you want to use a different job document, choose **Browse** and select a different bucket and document. Choose **Next**.

4. On the **Job configuration** page, choose the job type as continuous or a snapshot job. A snapshot job is complete when it finishes its run on the target devices and groups. A continuous job applies to thing groups and runs on any device that you add to a specified target group.

5. Continue to add any additional configurations for your job and then review and create your job. For information about the additional configurations, see:

   - Job rollout, scheduling, and abort configurations
   - Job execution timeout and retry configurations

> **Note**
>
> When a job created from a job template updates the existing parameters provided by the job template, those updated parameters will override the existing parameters provided by the job template for that job.

You can also create jobs from job templates with Fleet Hub web applications. For information about creating jobs in Fleet Hub, see Working with job templates in Fleet Hub for AWS IoT Device Management.

Delete a job template

To delete a job template, first go to the Job templates hub of the AWS IoT console and choose the **Custom templates** tab. Then, choose the job template you want to delete and choose **Next**.

> **Note**
>
> A deletion is permanent and the job template no longer appears on the **Custom templates** tab.

Create custom job templates by using the AWS CLI

This topic explains how to create, delete, and retrieve details about job templates by using the AWS CLI.

Create a job template from scratch

The following AWS CLI command shows how to create a job using a job document (*job-document.json*) stored in an S3 bucket and a role with permission to download files from Amazon S3 (*S3DownloadRole*).

```
aws iot create-job-template  \
      --job-template-id 010  \
      --description "My custom job template for updating the device firmware"
      --document-source https://s3.amazonaws.com/amzn-s3-demo-bucket/job-document.json
 \
      --timeout-config inProgressTimeoutInMinutes=100 \
      --job-executions-rollout-config "{ \"exponentialRate\": { \"baseRatePerMinute\":
 50, \"incrementFactor\": 2, \"rateIncreaseCriteria\": { \"numberOfNotifiedThings\":
 1000, \"numberOfSucceededThings\": 1000}}, \"maximumPerMinute\": 1000}" \
      --abort-config "{ \"criteriaList\": [ { \"action\": \"CANCEL\", \"failureType
\": \"FAILED\", \"minNumberOfExecutedThings\": 100, \"thresholdPercentage\": 20},
 { \"action\": \"CANCEL\", \"failureType\": \"TIMED_OUT\", \"minNumberOfExecutedThings
\": 200, \"thresholdPercentage\": 50}]}" \
      --presigned-url-config "{\"roleArn\":\"arn:aws:iam::123456789012:role/
S3DownloadRole\", \"expiresInSec\":3600}"
```

The optional `timeout-config` parameter specifies the amount of time each device has to finish running the job. The timer starts when the job execution status is set to `IN_PROGRESS`. If the job execution status isn't set to another terminal state before the time expires, it's set to `TIMED_OUT`.

The in-progress timer can't be updated and applies to all job launches for the job. Whenever a job launch remains in the `IN_PROGRESS` state for longer than this interval, the job launch fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

For more information about creating configurations about job rollouts and aborts, see Job rollout and abort configuration.

> **Note**
>
> Job documents that are specified as Amazon S3 files are retrieved at the time you create the job. If you change the contents of the Amazon S3 file you used as the source of your job document after you create the job, what is sent to the targets of the job doesn't change.

Create a job template from an existing job

The following AWS CLI command creates a job template by specifying the Amazon Resource Name (ARN) of an existing job. The new job template uses all of the configurations specified in the job. Optionally, you can change any of the configurations in the existing job by using any of the optional parameters.

```
aws iot create-job-template  \
      --job-arn arn:aws:iot:region:123456789012:job/job-name  \
      --timeout-config inProgressTimeoutInMinutes=100
```

Get details about a job template

The following AWS CLI command gets details about a specified job template.

```
aws iot describe-job-template \
      --job-template-id template-id
```

The command displays the following output.

```
{
   "abortConfig": {
      "criteriaList": [
         {
            "action": "string",
            "failureType": "string",
            "minNumberOfExecutedThings": number,
            "thresholdPercentage": number
```

```
         }
      ]
   },
   "createdAt": number,
   "description": "string",
   "document": "string",
   "documentSource": "string",
   "jobExecutionsRolloutConfig": {
      "exponentialRate": {
         "baseRatePerMinute": number,
         "incrementFactor": number,
         "rateIncreaseCriteria": {
            "numberOfNotifiedThings": number,
            "numberOfSucceededThings": number
         }
      },
      "maximumPerMinute": number
   },
   "jobTemplateArn": "string",
   "jobTemplateId": "string",
   "presignedUrlConfig": {
      "expiresInSec": number,
      "roleArn": "string"
   },
   "timeoutConfig": {
      "inProgressTimeoutInMinutes": number
   }
}
```

List job templates

The following AWS CLI command lists all of the job templates in your AWS account.

```
aws iot list-job-templates
```

The command displays the following output.

```
{
   "jobTemplates": [
```

```
        {
            "createdAt": number,
            "description": "string",
            "jobTemplateArn": "string",
            "jobTemplateId": "string"
        }
    ],
    "nextToken": "string"
}
```

To retrieve additional pages of results, use the value of the `nextToken` field.

Delete a job template

The following AWS CLI command deletes a specified job template.

```
aws iot delete-job-template \
      --job-template-id template-id
```

The command displays no output.

Create a job from a custom job template

The following AWS CLI command creates a job from a custom job template. It targets a device named `thingOne` and specifies the Amazon Resource Name (ARN) of the job template to use as the basis for the job. You can override advanced configurations, such as timeout and cancel configurations, by passing the associated parameters of the `create-job` command.

> **Warning**
>
> The `document-parameters` object must be used with the `create-job` command only when creating jobs from AWS managed templates. This object must not be used with custom job templates. For an example that shows how to create jobs using this parameter, see Create a job by using managed templates.

```
aws iot create-job \
```

```
    --targets arn:aws:iot:region:123456789012:thing/thingOne  \
    --job-template-arn arn:aws:iot:region:123456789012:jobtemplate/template-id
```

Job configurations

You can have the following additional configurations for each job that you deploy to the specified targets.

- **Rollout**: Defines how many devices receive the job document every minute.
- **Scheduling**: Schedules a job for a future date and time in addition to using recurring maintenance windows.
- **Abort**: Cancels a job in cases such as when some devices don't receive the job notification, or your devices report failure for their job executions.
- **Timeout**: If there isn't a response from your job targets within a certain duration after their job executions have started, the job can fail.
- **Retry**: Retries the job execution if your device reports failure when attempting to complete a job execution, or if your job execution times out.

By using these configurations, you can monitor the status of your job execution and avoid a bad update from being sent to an entire fleet.

Topics

- How job configurations work
- Specify additional configurations

How job configurations work

You use the rollout and abort configurations when you're deploying a job, and the timeout and retry configurations for job execution. The following sections show more information about how these configurations work.

Topics

- Job rollout, scheduling, and abort configurations
- Job execution timeout and retry configurations

Job rollout, scheduling, and abort configurations

You can use the job rollout, scheduling, and abort configurations to define how many devices receive the job document, schedule a job rollout, and determine the criteria for canceling a job.

Job rollout configuration

You can specify how quickly targets are notified of a pending job execution. You can also create a staged rollout to manage updates, reboots, and other operations. To specify how your targets are notified, use job rollout rates.

Job rollout rates

You can create a rollout configuration by using either a constant rollout rate or an exponential rollout rate. To specify the maximum number of job targets to inform per minute, use a constant rollout rate.

AWS IoT jobs can be deployed using exponential rollout rates as various criteria and thresholds are met. If the number of failed jobs matches a set of criteria that you specify, then you can cancel the job rollout. You set the job rollout rate criteria when you create a job by using the `JobExecutionsRolloutConfig` object. You also set the job abort criteria at job creation by using the `AbortConfig` object.

The following example shows how rollout rates work. For example, a job rollout with a base rate of 50 per minute, increment factor of 2, and number of notified and succeeded devices each as 1,000, would work as follows: The job will start at a rate of 50 job executions per minute and continue at that rate until either 1,000 things have received job execution notifications, or 1,000 successful job executions have occurred.

The following table illustrates how the rollout would proceed over the first four increments.

Rollout rate per minute	50	100	200	400
Number of notified devices or successful job executions to satisfy a rate increase	1,000	2,000	3,000	4,000

> **Note**
>
> If you're at your max concurrent limit of 500 Jobs (`isConcurrent = True`), then all active jobs will remain with a status of `IN-PROGRESS` and not roll out any new job executions

until the number of concurrent jobs is 499 or less (`isConcurrent = False`). This applies to snapshot and continuous jobs.

If `isConcurrent = True`, the job is currently rolling out job executions to all devices in your target group. If `isConcurrent = False`, the job has completed the rollout of all job executions to all devices in your target group. It will update its status state once all devices in your target group reach a terminal state, or a threshold percentage of your target group if you selected a job abort configuration. The Job level status states for `isConcurrent = True` and `isConcurrent = False` are both `IN_PROGRESS`.

For more information about active and concurrent job limits, see Active and concurrent job limits.

Job rollout rates for continuous jobs using dynamic thing groups

When you use a continuous job to roll out remote operations on your fleet, AWS IoT Jobs rolls out job executions for devices in your target thing group. For new devices that are added to the dynamic thing group, these job executions continue to roll out to those devices even after the job has been created.

The rollout configuration can control the rollout rates only for devices that are added to the group until job creation. After a job has been created, for any new devices, the job executions are created in near real time as soon as the devices join the target group.

Job scheduling configuration

You can schedule a continuous or snapshot job up to a year in advance using a pre-determined start time, end time, and end behavior for what will happen to each job execution upon reaching the end time. Additionally, you can create an optional recurring maintenance window with a flexible frequency, start time, and duration for continuous jobs to roll out a job document to all devices within the target group.

Job scheduling configurations

Start time

The start time of a scheduled job is the future date and time that job will begin rollout of the job document to all devices in the target group. Start time for a scheduled job applies to continuous jobs and snapshot jobs. When a scheduled job is initially created, it maintains a status state of `SCHEDULED`. Upon arriving at the `startTime` that you selected, it updates to `IN_PROGRESS` and begins the job document rollout. The `startTime` must be less than or equal to one year from the initial date and time that you created the scheduled job.

For more information on the syntax for `startTime` when using an API command or the AWS CLI, see Timestamp.

For a job with the optional scheduling configuration that takes place during a recurring maintenance window in a location observing daylight savings time (DST), the time will change by one hour when switching from DST to standard time and from standard time to DST.

> **Note**
>
> The time zone displayed in the AWS Management Console is your current system time zone. However, these time zones will be converted into UTC in the system.

End time

The end time of a scheduled job is the future date and time that the job will stop rollout of the job document to any remaining devices in the target group. End time for a scheduled job applies to continuous jobs and snapshot jobs. After a scheduled job arrives at the selected `endTime`, and all job executions have reached a terminal state, it updates its status state from `IN_PROGRESS` to `COMPLETED`. The `endTime` must be less than or equal to two years from the initial date and time that you created the scheduled job. The minimum duration between `startTime` and `endTime` is 30 minutes. Job execution retry attempts will occur until the job reaches the `endTime`, then the `endBehavior` will dictate how to proceed.

For more information on the syntax for `endTime` when using an API command or the AWS CLI, see Timestamp.

For a job with the optional scheduling configuration that takes place during a recurring maintenance window in a location observing daylight savings time (DST), the time will change by one hour when switching from DST to standard time and from standard time to DST.

> **Note**
>
> The time zone displayed in the AWS Management Console is your current system time zone. However, these time zones will be converted into UTC in the system.

End behavior

The end behavior of a scheduled job determines what happens to the job and all unfinished job executions when the job reaches the selected `endTime`.

The following lists the end behaviors that you can select from when creating the job or job template:

- `STOP_ROLLOUT`
  - `STOP_ROLLOUT` stops the rollout of the job document to all remaining devices in the target group for the job. Additionally, all `QUEUED` and `IN_PROGRESS` job executions will continue until they reach a terminal state. This is the default end behavior unless you select `CANCEL` or `FORCE_CANCEL`.
- `CANCEL`
  - `CANCEL` stops the rollout of the job document to all remaining devices in the target group for the job. Additionally, all `QUEUED` job executions will be cancelled while all `IN_PROGRESS` job executions will continue until they reach a terminal state.
- `FORCE_CANCEL`
  - `FORCE_CANCEL` stops the rollout of the job document to all remaining devices in the target group for the job. Additionally, all `QUEUED` and `IN_PROGRESS` job executions will be cancelled.

> **Note**
>
> To select an `endbehavior`, you must select an `endtime`

Max duration

The max duration of a scheduled job must be less than or equal to two years regardless of the `startTime` and `endTime`.

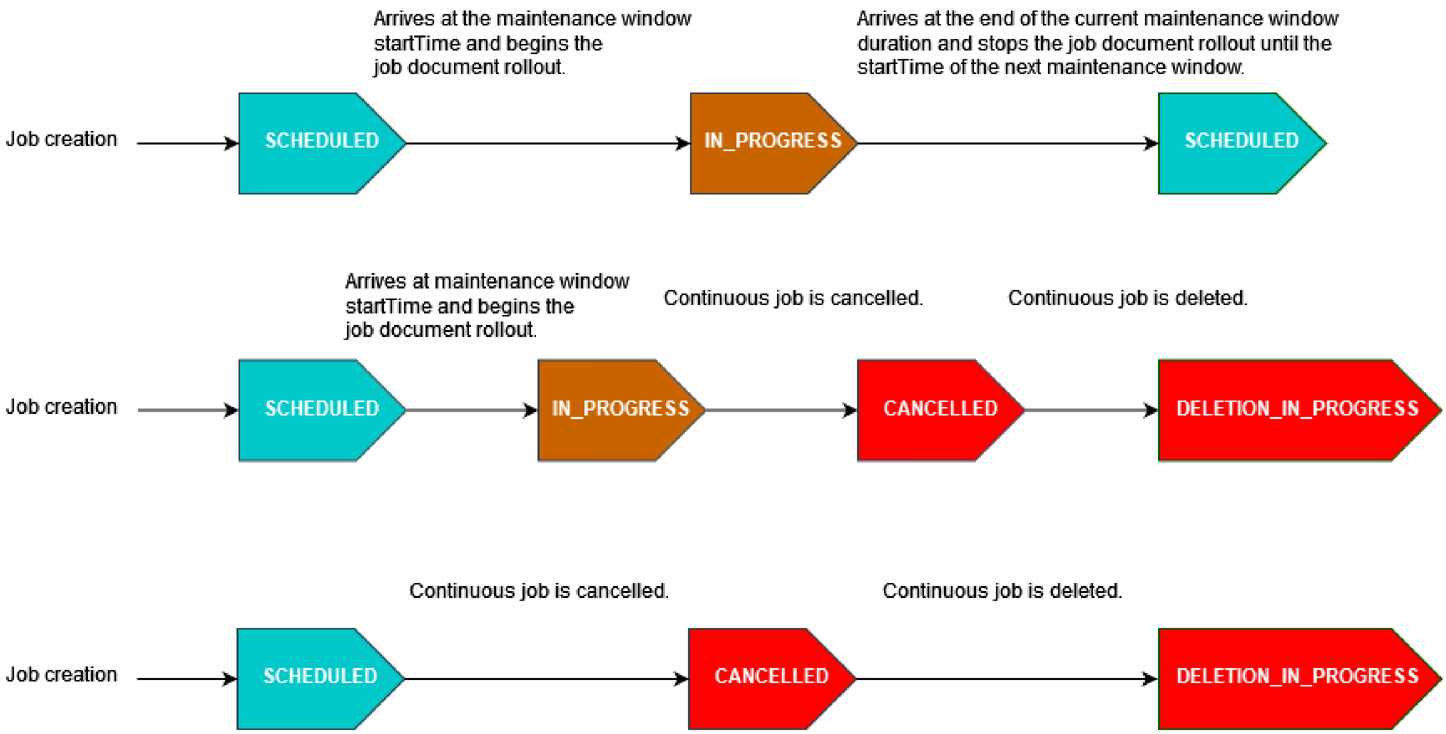
The following table lists common duration scenarios of a scheduled job:

Scheduled Job example number	startTime	endTime	Max duration
1	Immediately after initial job creation.	One year after initial job creation.	One year
2	One month after initial job creation.	13 months after initial job creation.	One year
3	One year after initial job creation.	Two years after initial job creation.	One year
4	Immediately after initial job creation.	Two years after initial job creation.	Two years

Recurring maintenance window

The maintenance window is an optional configuration within the scheduling configuration of the AWS Management Console and `SchedulingConfig` within the `CreateJob` and `CreateJobTemplate` APIs. You can set up a recurring maintenance window with a predetermined start time, duration, and frequency (daily, weekly, or monthly) that the maintenance window occurs. Maintenance windows only apply to continuous jobs. The maximum duration of a recurring maintenance window is 23 hours, 50 minutes.

The following diagram illustrates the job status states for various scheduled job scenarios with an optional maintenance window:



For more information about job status states, see Jobs and job execution states.

> **Note**
>
> If a job arrives at the `endTime` during a maintenance window, it will update from `IN_PROGRESS` to `COMPLETED`. Additionally, any remaining job executions will follow the `endBehavior` for the job.

Cron expressions

For scheduled jobs rolling out the job document during a maintenance window with a custom frequency, the custom frequency is entered using a cron expression. A cron expression has six required fields, which are separated by white space.

Syntax

```
cron(fields)
```

Field	Values	Wildcards
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W
Month	1-12 or JAN-DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? L #
Year	1970-2199	, - * /

Wildcards

- The **,** (comma) wildcard includes additional values. In the Month field, JAN,FEB,MAR would include January, February, and March.
- The **-** (dash) wildcard specifies ranges. In the Day field, 1-15 would include days 1 through 15 of the specified month.
- The ***** (asterisk) wildcard includes all values in the field. In the Hours field, ***** would include every hour. You can't use ***** in both the Day-of-month and Day-of-week fields. If you use it in one, you must use **?** in the other.
- The **/** (forward slash) wildcard specifies increments. In the Minutes field, you could enter 1/10 to specify every tenth minute, starting from the first minute of the hour (for example, the 11th, 21st, and 31st minute, and so on).
- The **?** (question mark) wildcard specifies one or another. In the Day-of-month field, you could enter **7** and if you didn't care what day of the week the 7th was, you could enter **?** in the Day-of-week field.
- The **L** wildcard in the Day-of-month or Day-of-week fields specifies the last day of the month or week.
- The `W` wildcard in the Day-of-month field specifies a weekday. In the Day-of-month field, `3W` specifies the weekday closest to the third day of the month.
- The **#** wildcard in the Day-of-week field specifies a certain instance of the specified day of the week within a month. For example, 3#2 would be the second Tuesday of the month: the 3 refers

to Tuesday because it is the third day of each week, and the 2 refers to the second day of that type within the month.

Note

If you use a '#' character, you can define only one expression in the day-of-week field. For example, `"3#1,6#3"` isn't valid because it's interpreted as two expressions.

Restrictions

- You can't specify the Day-of-month and Day-of-week fields in the same cron expression. If you specify a value (or a *) in one of the fields, you must use a **?**in the other.

Examples

Refer to the following sample cron strings when using a cron expression for the `startTime` of a recurring maintenance window.

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0	10	*	*	?	*	Run at 10:00 am (UTC) every day
15	12	*	*	?	*	Run at 12:15 pm (UTC) every day
0	18	?	*	MON-FRI	*	Run at 6:00 pm (UTC) every Monday

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
						through Friday
0	8	1	*	?	*	Run at 8:00 am (UTC) every first day of the month

Recurring maintenance window duration end logic

When a job rollout during a maintenance window reaches the end of the current maintenance window occurrence duration, the following actions will occur:

- The Job will cease all rollouts of the job document to any remaining devices in your target group. It will resume at the `startTime` of the next maintenance window.
- All job executions with a status of `QUEUED` will remain in `QUEUED` until the `startTime` of the next maintenance window occurrence. In the next window, they can switch to `IN_PROGRESS` when the device is ready to begin performing the actions specified in the job document.
- All job executions with a status of `IN_PROGRESS` will continue performing the actions specified in the job document until they reach a terminal state. Any retry attempts as specified in `JobExecutionsRetryConfig` will take place at the `startTime` of the next maintenance window.

Job abort configuration

Use this configuration to create a criteria to cancel a job when a threshold percentage of devices meet that criteria. For example, you can use this configuration to cancel a job in the following cases:

- When a threshold percentage of devices don't receive the job execution notifications, such as when your device is incompatible for an Over-The-Air (OTA) update. In this case, your device can report a `REJECTED` status.

- When a threshold percentage of devices report failure for their job executions, such as when your device encounters a disconnection when attempting to download the job document from an Amazon S3 URL. In such cases, your device must be programmed to report the `FAILURE` status to AWS IoT.
- When a `TIMED_OUT` status is reported because the job execution times out for a threshold percentage of devices after the job executions have started.
- When there are multiple retry failures. When you add a retry configuration, each retry attempt can incur additional charges to your AWS account. In such cases, canceling the job can cancel queued job executions and avoid retry attempts for these executions. For more information about the retry configuration and using it with the abort configuration, see Job execution timeout and retry configurations.

You can set up a job abort condition by using the AWS IoT console or the AWS IoT Jobs API.

Job execution timeout and retry configurations

Use the job execution timeout configuration to send you Jobs notifications when a job execution has been in progress for longer than the set duration. Use the job execution retry configuration to retry the execution when the job fails or times out.

Job execution timeout configuration

Use the job execution timeout configuration to notify you whenever a job execution gets stuck in the `IN_PROGRESS` state for an unexpectedly long period of time. When the job is `IN_PROGRESS`, you can monitor the progress of your job execution.

Timers for job timeouts

There are two types of timers: in-progress timers and step timers.

In-progress timers

When you create a job or a job template, you can specify a value for the in-progress timer that's between 1 minute and 7 days. You can update the value of this timer until the start of your job execution. After your timer starts, it can't be updated, and the timer value applies to all job executions for the job. Whenever a job execution remains in the `IN_PROGRESS` status for longer than this interval, the job execution fails and switches to the terminal `TIMED_OUT` status. AWS IoT also publishes an MQTT notification.

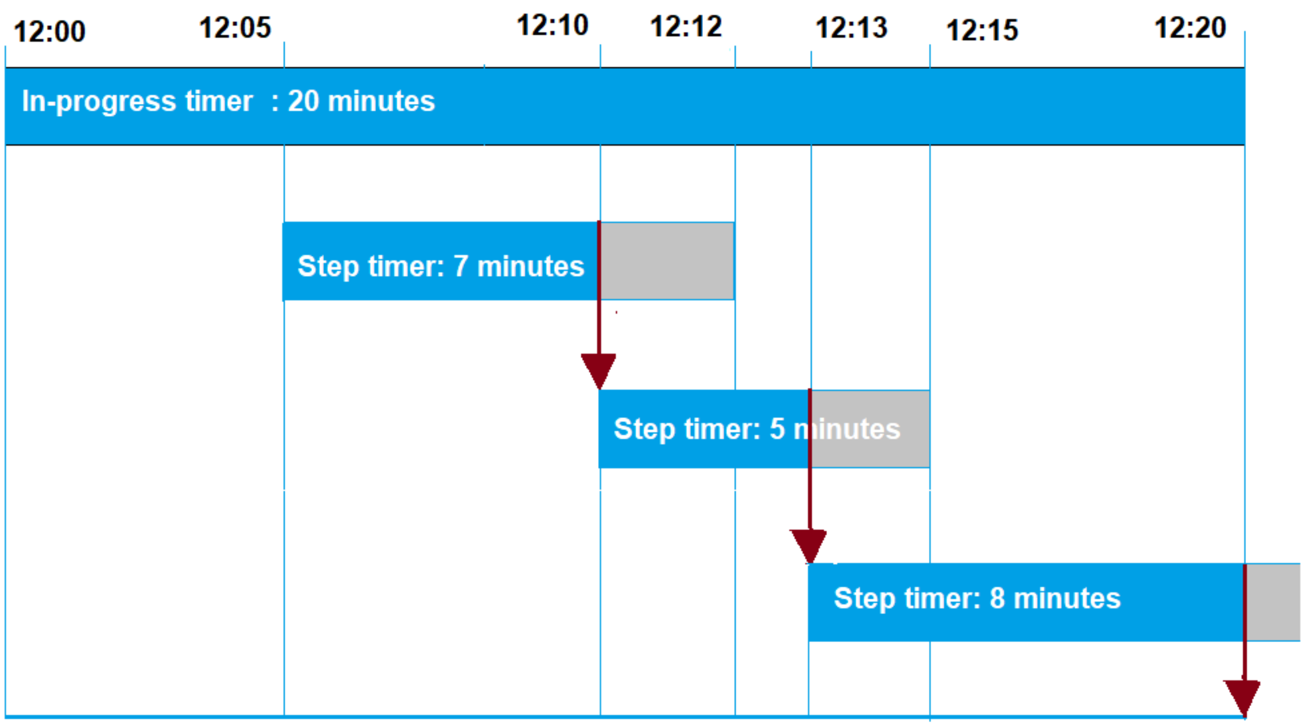
Step timer

You can also set a step timer that applies to only the job execution that you want to update. This timer has no effect on the in-progress timer. Each time you update a job execution, you can set a new value for the step timer. You can also create a new step timer when starting the next pending job execution for a thing. If the job execution remains in the `IN_PROGRESS` status for longer than the step timer interval, it fails and switches to the terminal `TIMED_OUT` status.

> **Note**
>
> You can set the in-progress timer by using the AWS IoT console or the AWS IoT Jobs API. To specify the step timer, use the API.

How timers work for job timeouts

The following illustrates the ways in which in-progress timeouts and step timeouts interact with each other in a 20-minute timeout period.



The following shows the different steps:

1. **12:00**

A new job is created and an in-progress timer for twenty minutes is started when creating a job. The in-progress timer starts to run and the job execution switches to `IN_PROGRESS` status.

2. **12:05 PM**

   A new step timer with a value of 7 minutes is created. The job execution will now time out at 12:12 PM.

3. **12:10 PM**

   A new step timer with a value of 5 minutes is created. When a new step timer is created, the previous step timer is discarded, and the job execution will now time out at 12:15 PM.

4. **12:13 PM**

   A new step timer with a value of 9 minutes is created. The previous step timer is discarded and the job execution will now time out at 12:20 PM because the in-progress timer times out at 12:20 PM. The step timer can't exceed the in-progress timer's absolute bound.

Job execution retry configuration

You can use the retry configuration to retry the job execution when a certain set of criteria is met. A retry can be attempted when a job times out or when the device fails. To retry execution because of a timeout failure, you must enable the timeout configuration.

How to use the retry configuration

Use the following steps to retry the configuration:

1. Determine whether to use the retry configuration for `FAILED`, `TIMED_OUT`, or both failure criteria. For the `TIMED_OUT` status, after the status is reported, AWS IoT Jobs automatically retries the job execution for the device.
2. For the `FAILED` status, check whether your job execution failure can be retried. If it's retryable, program your device to report a `FAILURE` status to AWS IoT. The following section describes more about retryable and non-retryable failures.
3. Specify the number of retries to use for each failure type by using the preceding information. For a single device, you can specify up to 10 retries for both failure types combined. The retry attempts stop automatically when an execution succeeds or when it reaches the specified number of attempts.

4. Add an abort configuration to cancel the job if there are repeated retry failures to avoid additional charges from being incurred with a large number of retry attempts.

> **Note**
>
> When a job reaches the end of a recurring maintenance window occurrence, all `IN_PROGRESS` job executions will continue performing actions identified in the job document until they reach a terminal state. If a job execution reaches a terminal state of `FAILED` or `TIMED_OUT` outside of a maintenance window, a retry attempt will occur in the next window if the attempts aren't exhausted. At the `startTime` of the next maintenance window occurrence, a new job execution will be created and enter a status state of `QUEUED` until the device is ready to begin.

Retry and abort configuration

Each retry attempt incurs additional charges to your AWS account. To avoid incurring additional charges from repeated retry failures, we recommend adding an abort configuration. For more information about pricing, see [AWS IoT Device Management pricing](https://aws.amazon.com/iot-device-management/pricing/).

You might encounter multiple retry failures when a high threshold percentage of your devices either time out or report failure. In this case, you can use the abort configuration to cancel the job and avoid any queued job executions or further retry attempts.

> **Note**
>
> When the abort criteria is met for canceling a job execution, only `QUEUED` job executions are canceled. Any queued retries for the device will not be attempted. However, current job executions that have an `IN_PROGRESS` status will not be canceled.

Before retrying a failed job execution, we also recommend that you check whether your job execution failure is retryable, as described in the following section.

Retry for failure type of `FAILED`

To attempt retries for failure type of `FAILED`, your devices must be programmed to report the `FAILURE` status for a failed job execution to AWS IoT. Set the retry configuration with the criteria to retry `FAILED` job executions and specify the number of retries to be performed. When AWS IoT

Jobs detects the FAILURE status, it will then automatically attempt to retry the job execution for the device. The retries continue until the job execution succeeds or when it reaches the maximum number of retry attempts.

You can track each retry attempt and the job that's running on these devices. By tracking the execution status, after the specified number of retries have been attempted, you can use your device to report failures and initiate another retry attempt.

Retryable and non-retryable failures

Your job execution failure can be retryable or non-retryable. Each retry attempt can incur charges to your AWS account. To avoid incurring additional charges from multiple retry attempts, first consider checking whether your job execution failure is retryable. An example of retryable failure includes a connection error that your device encounters while attempting to download the job document from an Amazon S3 URL. If your job execution failure is retryable, program your device to report a FAILURE status in case the job execution fails. Then, set the retry configuration to retry FAILED executions.

If the execution can't be retried, to avoid retrying and potentially incurring additional charges to your account, we recommend that you program the device to report a REJECTED status to AWS IoT. Examples of non-retryable failure include when your device is incompatible of receiving a job update, or when it experiences a memory error while executing a job. In these cases, AWS IoT Jobs will not retry the job execution because it retries the job execution only when it detects a FAILED or TIMED_OUT status.

After you've determined that a job execution failure is retryable, if a retry attempt still fails, consider checking the device logs.

> **Note**
>
> When a job with the optional scheduling configuration reaches its `endTime`, the selected `endBehavior` will stop the rollout of the job document to all remaining devices in the target group and dictate how to proceed with the remaining job executions. The attempts are retried if selected via the retry configuration.

Retry for failure type of `TIMEOUT`

If you enable timeout when creating a job, then AWS IoT Jobs will attempt to retry the job execution for the device when the status changes from IN_PROGRESS to TIMED_OUT. This status change can occur when the in-progress timer times out, or when a step timer that you specify is in `IN_PROGRESS` and then times out. The retries continue until the job execution succeeds, or when it reaches the maximum number of retry attempts for this failure type.

Continuous jobs and thing group membership updates

For continuous jobs that have a job status as `IN_PROGRESS`, the number of retry attempts is reset to zero when there are updates to a thing's group membership. For example, consider that you specified five retry attempts and three retries have already been performed. If a thing is now removed from the thing group and then rejoins the group, such as with dynamic thing groups, the number of retry attempts is reset to zero. You can now perform five retry attempts for your thing group instead of the two attempts that were remaining. In addition, when a thing is removed from the thing group, additional retry attempts are canceled.

Specify additional configurations

When you create a job or job template, you can specify these additional configurations. The following shows when you can specify these configurations.

- When creating a custom job template. The additional configuration settings that you specify will be saved when you create a job from the template.
- When creating a custom job by using a job file. The job file can be a JSON file that's uploaded in an S3 bucket.
- When creating a custom job by using a custom job template. If the template already has these settings specified, you can either reuse them or override them by specifying new configuration settings.
- When creating a custom job by using an AWS managed template.

Topics

Specify job configurations by using the AWS Management Console

You can add the different configurations for your job by using the AWS IoT console. After you've created a job, you can see the status details of your job configurations on the job details page.

For more information about the different configurations and how they work, see How job configurations work.

Add the job configurations when you create a job or a job template.

When creating a custom job template

To specify the rollout configuration when creating a custom job template

1. Go to the Job templates hub of the AWS IoT console and choose **Create job template**.
2. Specify the job template properties, provide the job document, expand the configuration that you want to add, and then specify the configuration parameters.

When creating a custom job

To specify the rollout configuration when creating a custom job

1. Go to the Job hub of the AWS IoT console and choose **Create job**.
2. Choose **Create a custom job** and specify the job properties, targets, and whether to use a job file or a template for the job document. You can use a custom template or an AWS managed template.
3. Choose the job configuration and then expand **Rollout configuration** to specify whether to use a **Constant rate** or **Exponential rate**. Then, specify the configuration parameters.

The next section shows the parameters that you can specify for each configuration.

Rollout configuration

You can specify whether to use a constant rollout rate or an exponential rate.

- **Set a constant rollout rate**

  To set a constant rate for job executions, choose **Constant rate**, then specify the **Maximum per minute** for the upper limit of the rate. This value is optional and ranges from 1 to 1000. If you don't set it, it uses 1000 as the default value.
- **Set an exponential rollout rate**

  To set an exponential rate, choose **Exponential rate** and then specify these parameters:

  - **Base rate per minute**

    The rate at which the jobs are executed until the **Number of notified devices** or **Number of succeeded devices** threshold is met for **Rate increase criteria**.
  - **Increment factor**

    The exponential factor by which the rollout rate increases after the **Number of notified devices** or **Number of succeeded devices** threshold is met for **Rate increase criteria**.
  - **Rate increase criteria**

    The threshold for either **Number of notified devices** or **Number of succeeded devices**.

Abort configuration

Choose **Add new configuration** and specify the following parameters for each configuration:

- **Failure type**

  Specifies the failure types that initiate a job abort. These include **FAILED**, **REJECTED**, **TIMED_OUT**, or **ALL**.
- **Increment factor**

  Specifies the number of completed job executions that must occur before the job abort criteria has been met.
- **Threshold percentage**

  Specifies the total number of executed things that initiate a job abort.

Scheduling configuration

Each job can start immediately upon initial creation, scheduled to start at a later date and time, or take place during a recurring maintenance window.

Choose **Add new configuration** and specify the following parameters for each configuration:

- **Job start**

  Specify the date and time when the job will start.
- **Recurring maintenance window**

A recurring maintenance window defines the specific date and time that a job can deploy the job document to the target devices in the job. The maintenance window can repeat daily, weekly, monthly, or a custom day and time recurrence.

- **Job end**

  Specify the date and time when the job will end.

- **Job end behavior**

  Select an end behavior for all unfinished job executions when the job is over.

> **Note**
>
> When a job with the optional scheduling configuration and selected end time reaches the end time, the job stops the rollout to all remaining devices in the target group. It also leverages the selected end behavior on how to proceed with the remaining job executions and their retry attempts per the retry configuration.

Timeout configuration

By default, there's no timeout and your job runs canceled or deleted. To use timeouts, choose **Enable timeout**, and then specify a timeout value between 1 minute and 7 days.

Retry configuration

> **Note**
>
> After a job has been created, the number of retries can't be updated. You can only remove the retry configuration for all failure types. When you're creating a job, consider the appropriate number of retries to use for your configuration. To avoid incurring excess costs because of potential retry failures, add an abort configuration.

Choose **Add new configuration** and specify the following parameters for each configuration:

- **Failure type**

Specifies the failure types that should trigger a job execution retry. These include **Failed**, **Timeout**, and **All**.

- **Number of retries**

  Specifies the number of retries for the chosen **Failure type**. For both failure types combined, up to 10 retries can be attempted.

Specify job configurations by using the AWS IoT Jobs API

You can use the CreateJob or the CreateJobTemplate API to specify the different job configurations. The following sections describe how to add these configurations. After you've added the configurations, you can use JobExecutionSummary and JobExecutionSummaryForJob to view their status.

For more information about the different configurations and how they work, see How job configurations work.

Rollout configuration

You can specify a constant rollout rate or an exponential rollout rate for your rollout configuration.

- **Set a constant rollout rate**

  To set a constant rollout rate, use the `JobExecutionsRolloutConfig` object to add the `maximumPerMinute` parameter to the `CreateJob` request. This parameter specifies the upper limit of the rate at which job executions can occur. This value is optional and ranges from 1 to 1000. If you don't set the value, it uses 1000 as the default value.

  ```
  "jobExecutionsRolloutConfig": {
      "maximumPerMinute": 1000
  }
  ```

- **Set an exponential rollout rate**

  To set a variable job rollout rate, use the `JobExecutionsRolloutConfig` object. You can configure the `ExponentialRolloutRate`property when you run the `CreateJob` API operation. The following example sets an exponential rollout rate by using the `exponentialRate` parameter. For more information about the parameters, see `ExponentialRolloutRate`.

```
{
...
  "jobExecutionsRolloutConfig": {
    "exponentialRate": {
      "baseRatePerMinute": 50,
      "incrementFactor": 2,
      "rateIncreaseCriteria": {
        "numberOfNotifiedThings": 1000,
        "numberOfSucceededThings": 1000
      },
      "maximumPerMinute": 1000
    }
  }
...
}
```

Where the parameter:

baseRatePerMinute

Specifies the rate at which the jobs are executed until the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold has been met.

incrementFactor

Specifies the exponential factor by which the rollout rate increases after the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold has been met.

rateIncreaseCriteria

Specifies either the `numberOfNotifiedThings` or `numberOfSucceededThings` threshold.

Abort configuration

To add this configuration by using the API, specify the [`AbortConfig`] parameter when you run the [`CreateJob`], or the [`CreateJobTemplate`] API operation. The following example shows an abort configuration for a job rollout that was experiencing multiple failed executions, as specified with the `CreateJob` API operation.

> **Note**
>
> Deleting a job execution affects the computation value of the total completed execution. When a job aborts, the service creates an automated `comment` and `reasonCode` to differentiate a user-driven cancellation from a job abort cancellation.

```
"abortConfig": {
   "criteriaList": [
      {
         "action": "CANCEL",
         "failureType": "FAILED",
         "minNumberOfExecutedThings": 100,
         "thresholdPercentage": 20
      },
      {
         "action": "CANCEL",
         "failureType": "TIMED_OUT",
         "minNumberOfExecutedThings": 200,
         "thresholdPercentage": 50
      }
   ]
 }
```

Where the parameter:

action

Specifies the action to take when the abort criteria has been met. This parameter is required, and `CANCEL` is the only valid value.

failureType

Specifies which failure types should initiate a job abort. Valid values are `FAILED`, `REJECTED`, `TIMED_OUT`, and `ALL`.

minNumberOfExecutedThings

Specifies the number of completed job executions that must occur before the job abort criteria has been met. In this example, AWS IoT doesn't check to see if a job abort should occur until at least 100 devices have completed job executions.

thresholdPercentage

Specifies the total number of things for which jobs are executed that can initiate a job abort. In this example, AWS IoT checks sequentially and initiates a job abort if the threshold percentage is met. If at least 20% of the complete executions failed after 100 executions are complete, it cancels the job rollout. If this criteria isn't met, AWS IoT then checks if at least 50% of completed executions timed out after 200 executions are complete. If this is the case, it cancels the job rollout.

Scheduling configuration

To add this configuration by using the API, specify the optional `SchedulingConfig` when you run the `CreateJob`, or the `CreateJobTemplate` API operation.

```
 "SchedulingConfig": {
   "endBehavior": string
   "endTime": string
   "maintenanceWindows": string
   "startTime": string
}
```

Where the parameter:

startTime

Specifies the date and time when the job will start.

endTime

Specifies the date and time when the job will end.

maintenanceWindows

Specifies if an optional maintenance window was selected for the scheduled job to rollout the job document to all devices in the target group. The string format for `maintenanceWindow` is YYYY/MM/DD for the date and hh:mm for the time.

endBehavior

Specifies the job behavior for a scheduled job upon reaching the `endTime`.

> **Note**
>
> The optional `SchedulingConfig` for a job is viewable in the `DescribeJob` and `DescribeJobTemplate` APIs.

Timeout configuration

To add this configuration by using the API, specify the `TimeoutConfig` parameter when you run the `CreateJob`, or the `CreateJobTemplate` API operation.

To use the timeout configuration

1. To set the in-progress timer when you're creating a job or job template, set a value for the `inProgressTimeoutInMinutes` property of the optional TimeoutConfig object.

```
    "timeoutConfig": {
      "inProgressTimeoutInMinutes": number
   }
```

2. To specify a step timer for a job execution, set a value for `stepTimeoutInMinutes` when you call UpdateJobExecution. The step timer applies only to the job execution that you update. You can set a new value for this timer each time you update a job execution.

> **Note**
>
> `UpdateJobExecution` can discard a step timer that's already been created by creating a new step timer with a value of -1.

```
{
   ...
    "statusDetails": {
      "string" : "string"
   },
   "stepTimeoutInMinutes": number
}
```

3. To create a new step timer, you can also call the StartNextPendingJobExecution API operation.

Retry configuration

Note

When you're creating a job, consider the appropriate number of retries to use for your configuration. To avoid incurring excess costs because of potential retry failures, add an abort configuration. After a job has been created, the number of retries can't be updated. You can only set the number of retries to 0 by using the UpdateJob API operation.

To add this configuration by using the API, specify the `jobExecutionsRetryConfig` parameter when you run the `CreateJob`, or the `CreateJobTemplate` API operation.

```
{
...
  "jobExecutionsRetryConfig": {
      "criteriaList": [
         {
            "failureType": "string",
            "numberOfRetries": number
         }
      ]
  }
...
}
```

Where **criteriaList** is an array specifying the list of criteria that determines the number of retries permitted for each failure type for a job.

Devices and jobs

Devices can communicate with AWS IoT Jobs using MQTT, HTTP Signature Version 4, or HTTP TLS. To determine the endpoint to use when your device communicates with AWS IoT Jobs, run the **DescribeEndpoint** command. For example, if you run this command:

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

you get a result similar to the following:

```
{
    "endpointAddress": "a1b2c3d4e5f6g7-ats.iot.us-west-2.amazonaws.com"
}
```

Using the MQTT protocol

Devices can communicate with AWS IoT Jobs using MQTT protocol. Devices subscribe to MQTT topics to be notified of new jobs and to receive responses from the AWS IoT Jobs service. Devices publish on MQTT topics to query or update the state of a job launch. Each device has its own general MQTT topic. For more information about publishing and subscribing to MQTT topics, see Device communication protocols.

With this method of communication, your device uses its device-specific certificate and private key to authenticate with AWS IoT Jobs.

Your devices can subscribe to the following topics. `thing-name` is the name of the thing associated with the device.

- **`$aws/things/`*`thing-name`*`/jobs/notify`**

  Subscribe to this topic to notify you when a job launch is added or removed from the list of pending job launches.
- **`$aws/things/`*`thing-name`*`/jobs/notify-next`**

  Subscribe to this topic to notify you when the next pending job execution has changed.
- **`$aws/things/`*`thing-name`*`/jobs/`*`request-name`*`/accepted`**

  The AWS IoT Jobs service publishes success and failure messages on an MQTT topic. The topic is formed by appending `accepted` or `rejected` to the topic used to make the request. Here, `request-name` is the name of a request such as `Get` and the topic can be: `$aws/things/myThing/jobs/get`. AWS IoT Jobs then publishes success messages on the `$aws/things/myThing/jobs/get/accepted` topic.
- **`$aws/things/`*`thing-name`*`/jobs/`*`request-name`*`/rejected`**

  Here, `request-name` is the name of a request such as `Get`. If the request failed, AWS IoT Jobs publishes failure messages on the `$aws/things/myThing/jobs/get/rejected` topic.

You can also use the following HTTPS API operations:

- Update the status of a job execution by calling the `UpdateJobExecution` API.
- Query the status of a job execution by calling the `DescribeJobExecution` API.
- Retrieve a list of pending job executions by calling the `GetPendingJobExecutions` API.
- Retrieve the next pending job execution by calling the `DescribeJobExecution` API with `jobId` as `$next`.
- Get and start the next pending job execution by calling the `StartNextPendingJobExecution` API.

Using HTTP Signature Version 4

Devices can communicate with AWS IoT Jobs using HTTP Signature Version 4 on port 443. This is the method used by the AWS SDKs and CLI. For more information about those tools, see AWS CLI Command Reference: iot-jobs-data or AWS SDKs and Tools and refer to the IotJobsDataPlane section for your preferred language.

With this method of communication, your device uses IAM credentials to authenticate with AWS IoT Jobs.

The following commands are available using this method:

- **DescribeJobExecution**

```
aws iot-jobs-data describe-job-execution ...
```

- **GetPendingJobExecutions**

```
aws iot-jobs-data get-pending-job-executions ...
```

- **StartNextPendingJobExecution**

```
aws iot-jobs-data start-next-pending-job-execution ...
```

- **UpdateJobExecution**

```
aws iot-jobs-data update-job-execution ...
```

Using HTTP TLS

Devices can communicate with AWS IoT Jobs using HTTP TLS on port 8443 using a third-party software client that supports this protocol.

With this method, your device uses X.509 certificate-based authentication (for example, its device-specific certificate and private key).

The following commands are available using this method:

- **DescribeJobExecution**
- **GetPendingJobExecutions**
- **StartNextPendingJobExecution**
- **UpdateJobExecution**

Programming devices to work with jobs

The examples in this section use MQTT to illustrate how a device works with the AWS IoT Jobs service. Or, you could use the corresponding API or CLI commands. For these examples, we assume a device called `MyThing` that subscribes to the following MQTT topics:

- `$aws/things/MyThing/jobs/notify` (or `$aws/things/MyThing/jobs/notify-next`)
- `$aws/things/MyThing/jobs/get/accepted`
- `$aws/things/MyThing/jobs/get/rejected`
- `$aws/things/MyThing/jobs/jobId/get/accepted`
- `$aws/things/MyThing/jobs/jobId/get/rejected`

If you're using code signing for AWS IoT, your device code must verify the signature of your code file. The signature is in the job document in the `codesign` property. For more information about verifying a code file signature, see Device Agent Sample.

Topics

- Device workflow
- Jobs workflow
- Jobs notifications

Device workflow

A device can handle jobs that it runs using either of the following ways.

- **Get the next job**

1. When a device first comes online, it should subscribe to the device's `notify-next` topic.
2. Call the DescribeJobExecution MQTT API with jobId `$next` to get the next job, its job document, and other details, including any state saved in `statusDetails`. If the job document has a code file signature, you must verify the signature before proceeding with processing the job request.
3. Call the UpdateJobExecution MQTT API to update the job status. Or, to combine this and the previous step in one call, the device can call StartNextPendingJobExecution.
4. (Optional) You can add a step timer by setting a value for `stepTimeoutInMinutes` when you call either UpdateJobExecution or StartNextPendingJobExecution.
5. Perform the actions specified by the job document using the UpdateJobExecution MQTT API to report on the progress of the job.
6. Continue to monitor the job execution by calling the DescribeJobExecution MQTT API with this jobId. If the job execution is deleted, DescribeJobExecution returns a `ResourceNotFoundException`.

   The device should be able to recover to a valid state if the job execution is canceled or deleted while the device is running the job.
7. Call the UpdateJobExecution MQTT API when finished with the job to update the job status and report success or failure.
8. Because this job's execution status has been changed to a terminal state, the next job available for execution (if any) changes. The device is notified that the next pending job execution has changed. At this point, the device should continue as described in step 2.

If the device remains online, it continues to receive notifications of the next pending job execution. This includes its job execution data, when it completes a job or a new pending job execution is added. When this occurs, the device continues as described in step 2.

- **Select from available jobs**

1. When a device first comes online, it should subscribe to the thing's `notify` topic.
2. Call the GetPendingJobExecutions MQTT API to get a list of pending job executions.
3. If the list contains one or more job executions, select one.
4. Call the DescribeJobExecution MQTT API to get the job document and other details, including any state saved in `statusDetails`.

5. Call the UpdateJobExecution MQTT API to update the job status. If the `includeJobDocument` field is set to `true` in this command, the device can skip the previous step and retrieve the job document at this point.
6. Optionally, you can add a step timer by setting a value for `stepTimeoutInMinutes` when you call UpdateJobExecution.
7. Perform the actions specified by the job document using the UpdateJobExecution MQTT API to report on the progress of the job.
8. Continue to monitor the job execution by calling the DescribeJobExecution MQTT API with this jobId. If the job execution is canceled or deleted while the device is running the job, the device should be able to recover to a valid state.
9. Call the UpdateJobExecution MQTT API when finished with the job to update the job status and to report success or failure.

If the device remains online, it is notified of all pending job executions when a new pending job execution becomes available. When this occurs, the device can continue as described in step 2.

If the device is unable to carry out the job, it should call the UpdateJobExecution MQTT API to update the job status to `REJECTED`.

Jobs workflow

The following shows the different steps in the jobs workflow from starting a new job to reporting the completion status of a job execution.

Start a new job

When a new job is created, AWS IoT Jobs publishes a message on the `$aws/things/`*`thing-name`*`/jobs/notify` topic for each target device.

The message contains the following information:

```
{
    "timestamp":1476214217017,
    "jobs":{
        "QUEUED":[{
            "jobId":"0001",
            "queuedAt":1476214216981,
```

```
            "lastUpdatedAt":1476214216981,
            "versionNumber" : 1
        }]
    }
}
```

The device receives this message on the '$aws/things/*thingName*/jobs/notify' topic when the job execution is queued.

> **Note**
>
> For jobs with the optional `SchedulingConfig`, the job will maintain an initial status state of `SCHEDULED`. When the job reaches the selected `startTime`, the following will occur:
>
> - The job status state will update to `IN_PROGRESS`.
> - The job will begin rollout of the job document to all devices in the target group.

Get job information

To get more information about a job execution, the device calls the DescribeJobExecution MQTT API with the `includeJobDocument` field set to `true` (the default).

If the request is successful, the AWS IoT Jobs service publishes a message on the `$aws/things/MyThing/jobs/0023/get/accepted` topic:

```
{
    "clientToken" : "client-001",
    "timestamp" : 1489097434407,
    "execution" : {
        "approximateSecondsBeforeTimedOut": number,
        "jobId" : "023",
        "status" : "QUEUED",
        "queuedAt" : 1489097374841,
        "lastUpdatedAt" : 1489097374841,
        "versionNumber" : 1,
        "jobDocument" : {
            < contents of job document >
        }
    }
}
```

If the request fails, the AWS IoT Jobs service publishes a message on the `$aws/things/MyThing/jobs/0023/get/rejected` topic.

The device now has the job document that it can use to perform the remote operations for the job. If the job document contains an Amazon S3 presigned URL, the device can use that URL to download any required files for the job.

Report job execution status

As the device is executing the job, it can call the UpdateJobExecution MQTT API to update the status of the job execution.

For example, a device can update the job execution status to `IN_PROGRESS` by publishing the following message on the `$aws/things/MyThing/jobs/0023/update` topic:

```
{
    "status":"IN_PROGRESS",
    "statusDetails": {
        "progress":"50%"
    },
    "expectedVersion":"1",
    "clientToken":"client001"
}
```

Jobs respond by publishing a message to the `$aws/things/MyThing/jobs/0023/update/accepted` or `$aws/things/MyThing/jobs/0023/update/rejected` topic:

```
{
    "clientToken":"client001",
    "timestamp":1476289222841
}
```

The device can combine the two previous requests by calling StartNextPendingJobExecution. That gets and starts the next pending job execution and allows the device to update the job execution status. This request also returns the job document when there is a job execution pending.

If the job contains a TimeoutConfig, the in-progress timer starts running. You can also set a step timer for a job execution by setting a value for `stepTimeoutInMinutes` when you call UpdateJobExecution. The step timer applies only to the job execution that you update. You can set a new value for this timer each time you update a job execution. You can also create a step timer when you call StartNextPendingJobExecution. If the job execution remains in the `IN_PROGRESS` status for longer than the step timer interval, it fails and switches to the terminal TIMED_OUT status. The step timer has no effect on the in-progress timer that you set when you create a job.

The status field can be set to IN_PROGRESS, SUCCEEDED, or FAILED. You cannot update the status of a job execution that is already in a terminal state.

Report execution completed

When the device is finished executing the job, it calls the UpdateJobExecution MQTT API. If the job was successful, set status to SUCCEEDED and, in the message payload, in statusDetails, add other information about the job as name-value pairs. The in-progress and step timers end when the job execution is complete.

For example:

```
{
    "status":"SUCCEEDED",
    "statusDetails": {
        "progress":"100%"
    },
    "expectedVersion":"2",
    "clientToken":"client-001"
}
```

If the job was not successful, set status to FAILED and, in statusDetails, add information about the error that occurred:

```
{
    "status":"FAILED",
    "statusDetails": {
        "errorCode":"101",
        "errorMsg":"Unable to install update"
    },
    "expectedVersion":"2",
    "clientToken":"client-001"
}
```

> **Note**
>
> The statusDetails attribute can contain any number of name-value pairs.

When the AWS IoT Jobs service receives this update, it publishes a message on the `$aws/things/MyThing/jobs/notify` topic to indicate that the job execution is complete:

```
{
    "timestamp":1476290692776,
    "jobs":{}
}
```

Additional jobs

If there are other job executions pending for the device, they are included in the message published to `$aws/things/MyThing/jobs/notify`.

For example:

```
{
    "timestamp":1476290692776,
    "jobs":{
        "QUEUED":[{
            "jobId":"0002",
            "queuedAt":1476290646230,
            "lastUpdatedAt":1476290646230
        }],
        "IN_PROGRESS":[{
            "jobId":"0003",
            "queuedAt":1476290646230,
            "lastUpdatedAt":1476290646230
        }]
    }
}
```

Jobs notifications

The AWS IoT Jobs service publishes MQTT messages to reserved topics when jobs are pending or when the first job execution in the list changes. Devices can track pending jobs by subscribing to these topics.

Job notification types

Job notifications are published to MQTT topics as JSON payloads. There are two kinds of notifications:

ListNotification

A `ListNotification` contains a list of no more than 15 pending job executions. They are sorted by status (`IN_PROGRESS` job executions before `QUEUED` job executions) and then by the times when they were queued.

A `ListNotification` is published whenever one of the following criteria is met.

- A new job execution is queued or changes to a non-terminal status (`IN_PROGRESS` or `QUEUED`).
- An old job execution changes to a terminal status (`FAILED`, `SUCCEEDED`, `CANCELED`, `TIMED_OUT`, `REJECTED`, or `REMOVED`).

For more information about the limits with and without the scheduling configuration, see [Job executions limits](#).

NextNotification

- A `NextNotification` contains summary information about the job execution that's next in the queue.

  A `NextNotification` is published whenever the first job execution in the list changes.
  - A new job execution is added to the list as `QUEUED`, and it's the first one in the list.
  - The status of an existing job execution that wasn't the first one in the list changes from `QUEUED` to `IN_PROGRESS`, and becomes the first one in the list. (This happens when there are no other `IN_PROGRESS` job executions in the list or when the job execution whose status changes from `QUEUED` to `IN_PROGRESS` was queued earlier than any other `IN_PROGRESS` job execution in the list.)
  - The status of the job execution that is first in the list changes to a terminal status and is removed from the list.

For more information about publishing and subscribing to MQTT topics, see [the section called “Device communication protocols”](#).

> **Note**
>
> Notifications are not available when you use HTTP Signature Version 4 or HTTP TLS to communicate with jobs.

Job pending

The AWS IoT Jobs service publishes a message on an MQTT topic when a job is added to or removed from the list of pending job executions for a thing or the first job execution in the list changes:

- `$aws/things/thingName/jobs/notify`
- `$aws/things/thingName/jobs/notify-next`

The messages contain the following example payloads:

`$aws/things/thingName/jobs/notify`:

```
{
  "timestamp" : 10011,
  "jobs" : {
    "IN_PROGRESS" : [ {
      "jobId" : "other-job",
      "queuedAt" : 10003,
      "lastUpdatedAt" : 10009,
      "executionNumber" : 1,
      "versionNumber" : 1
    } ],
    "QUEUED" : [ {
      "jobId" : "this-job",
      "queuedAt" : 10011,
      "lastUpdatedAt" : 10011,
      "executionNumber" : 1,
      "versionNumber" : 0
    } ]
  }
}
```

If the job execution called `this-job` originated from a job with the optional scheduling configuration selected and the job document rollout scheduled to take place during a maintenance window, it'll only appear during a recurring maintenance window. Outside of a maintenance window, the job called `this-job` will be excluded from the list of pending job executions as shown in the following example.

```
{
  "timestamp" : 10011,
```

```
  "jobs" : {
    "IN_PROGRESS" : [ {
      "jobId" : "other-job",
      "queuedAt" : 10003,
      "lastUpdatedAt" : 10009,
      "executionNumber" : 1,
      "versionNumber" : 1
    } ],
    "QUEUED" : []
  }
}
```

`$aws/things/thingName/jobs/notify-next`:

```
{
  "timestamp" : 10011,
  "execution" : {
    "jobId" : "other-job",
    "status" : "IN_PROGRESS",
    "queuedAt" : 10009,
    "lastUpdatedAt" : 10009,
    "versionNumber" : 1,
    "executionNumber" : 1,
    "jobDocument" : {"c":"d"}
  }
}
```

If the job execution called `other-job` originated from a job with the optional scheduling configuration selected and the job document rollout scheduled to take place during a maintenance window, it'll only appear during a recurring maintenance window. Outside of a maintenance window, the job called `other-job` won't be listed as the next job execution as shown in the following example.

```
{} //No other pending jobs
```

```
{
  "timestamp" : 10011,
  "execution" : {
      "jobId" : "this-job",
      "queuedAt" : 10011,
      "lastUpdatedAt" : 10011,
```

```
        "executionNumber" : 1,
        "versionNumber" : 0,
        "jobDocument" : {"a":"b"}
    }
} // "this-job" is pending next to "other-job"
```

Possible job execution status values are `QUEUED`, `IN_PROGRESS`, `FAILED`, `SUCCEEDED`, `CANCELED`, `TIMED_OUT`, `REJECTED`, and `REMOVED`.

The following series of examples show the published notifications to each topic as job executions are created and changed from one state to another.

First, one job, called `job1`, is created. This notification is published to the `jobs/notify` topic:

```
{
  "timestamp": 1517016948,
  "jobs": {
    "QUEUED": [
      {
        "jobId": "job1",
        "queuedAt": 1517016947,
        "lastUpdatedAt": 1517016947,
        "executionNumber": 1,
        "versionNumber": 1
      }
    ]
  }
}
```

This notification is published to the `jobs/notify-next` topic:

```
{
  "timestamp": 1517016948,
  "execution": {
    "jobId": "job1",
    "status": "QUEUED",
    "queuedAt": 1517016947,
    "lastUpdatedAt": 1517016947,
    "versionNumber": 1,
    "executionNumber": 1,
    "jobDocument": {
      "operation": "test"
    }
```

```
  }
}
```

When another job is created (`job2`), this notification is published to the `jobs/notify` topic:


```
{
  "timestamp": 1517017192,
  "jobs": {
    "QUEUED": [
      {
        "jobId": "job1",
        "queuedAt": 1517016947,
        "lastUpdatedAt": 1517016947,
        "executionNumber": 1,
        "versionNumber": 1
      },
      {
        "jobId": "job2",
        "queuedAt": 1517017191,
        "lastUpdatedAt": 1517017191,
        "executionNumber": 1,
        "versionNumber": 1
      }
    ]
  }
}
```


A notification is not published to the `jobs/notify-next` topic because the next job in the queue (`job1`) has not changed. When `job1` starts to execute, its status changes to `IN_PROGRESS`. No notifications are published because the list of jobs and the next job in the queue have not changed.

When a third job (`job3`) is added, this notification is published to the `jobs/notify` topic:


```
{
  "timestamp": 1517017906,
  "jobs": {
    "IN_PROGRESS": [
      {
        "jobId": "job1",
        "queuedAt": 1517016947,
        "lastUpdatedAt": 1517017472,
        "startedAt": 1517017472,
        "executionNumber": 1,
```

```
          "versionNumber": 2
        }
      ],
      "QUEUED": [
        {
          "jobId": "job2",
          "queuedAt": 1517017191,
          "lastUpdatedAt": 1517017191,
          "executionNumber": 1,
          "versionNumber": 1
        },
        {
          "jobId": "job3",
          "queuedAt": 1517017905,
          "lastUpdatedAt": 1517017905,
          "executionNumber": 1,
          "versionNumber": 1
        }
      ]
    }
}
```

A notification is not published to the `jobs/notify-next` topic because the next job in the queue is still `job1`.

When `job1` is complete, its status changes to `SUCCEEDED`, and this notification is published to the `jobs/notify` topic:

```
{
  "timestamp": 1517186269,
  "jobs": {
    "QUEUED": [
      {
        "jobId": "job2",
        "queuedAt": 1517017191,
        "lastUpdatedAt": 1517017191,
        "executionNumber": 1,
        "versionNumber": 1
      },
      {
        "jobId": "job3",
        "queuedAt": 1517017905,
        "lastUpdatedAt": 1517017905,
```

```
        "executionNumber": 1,
        "versionNumber": 1
      }
    ]
  }
}
```


At this point, `job1` has been removed from the queue, and the next job to be executed is `job2`. This notification is published to the `jobs/notify-next` topic:


```
{
  "timestamp": 1517186269,
  "execution": {
    "jobId": "job2",
    "status": "QUEUED",
    "queuedAt": 1517017191,
    "lastUpdatedAt": 1517017191,
    "versionNumber": 1,
    "executionNumber": 1,
    "jobDocument": {
      "operation": "test"
    }
  }
}
```


If `job3` must begin executing before `job2` (which is not recommended), the status of `job3` can be changed to `IN_PROGRESS`. If this happens, `job2` is no longer next in the queue, and this notification is published to the `jobs/notify-next` topic:


```
{
  "timestamp": 1517186779,
  "execution": {
    "jobId": "job3",
    "status": "IN_PROGRESS",
    "queuedAt": 1517017905,
    "startedAt": 1517186779,
    "lastUpdatedAt": 1517186779,
    "versionNumber": 2,
    "executionNumber": 1,
    "jobDocument": {
      "operation": "test"
    }
```

```
  }
}
```

No notification is published to the `jobs/notify` topic because no job has been added or removed.

If the device rejects `job2` and updates its status to `REJECTED`, this notification is published to the `jobs/notify` topic:

```
{
  "timestamp": 1517189392,
  "jobs": {
    "IN_PROGRESS": [
      {
        "jobId": "job3",
        "queuedAt": 1517017905,
        "lastUpdatedAt": 1517186779,
        "startedAt": 1517186779,
        "executionNumber": 1,
        "versionNumber": 2
      }
    ]
  }
}
```

If `job3` (which is still in progress) is force deleted, this notification is published to the `jobs/notify` topic:

```
{
  "timestamp": 1517189551,
  "jobs": {}
}
```

At this point, the queue is empty. This notification is published to the `jobs/notify-next` topic:

```
{
  "timestamp": 1517189551
}
```

AWS IoT jobs API operations

AWS IoT Jobs API can be used for either of the following categories:

- Administrative tasks such as management and control of jobs. This is the *control plane*.
- Devices carrying out those jobs. This is the *data plane*, which permits you to send and receive data.

Job management and control uses an HTTPS protocol API. Devices can use either an MQTT or an HTTPS protocol API. The control plane API is designed for a low volume of calls typical when creating and tracking jobs. It usually opens a connection for a single request, and then closes the connection after the response is received. The data plane HTTPS and MQTT API permit long polling. These API operations are designed for large amounts of traffic that can scale to millions of devices.

Each AWS IoT Jobs HTTPS API has a corresponding command that permits you to call the API from the AWS Command Line Interface (AWS CLI). The commands are lowercase, with hyphens between the words that make up the name of the API. For example, you can invoke the `CreateJob` API on the CLI by typing:

```
aws iot create-job ...
```

If an error occurs during an operation, you get an error response that contains information about the error.

ErrorResponse

Contains information about an error that occurred during an AWS IoT Jobs service operation.

The following example shows the syntax of this operation:

```
{
    "code": "ErrorCode",
    "message": "string",
    "clientToken": "string",
    "timestamp": timestamp,
    "executionState": JobExecutionState
}
```

The following is a description of this `ErrorResponse`:

`code`

ErrorCode can be set to:

InvalidTopic

The request was sent to a topic in the AWS IoT Jobs namespace that doesn't map to any API operation.

InvalidJson

The contents of the request couldn't be interpreted as valid UTF-8-encoded JSON.

InvalidRequest

The contents of the request were not valid. For example, this code is returned when an `UpdateJobExecution` request contains invalid status details. The message contains details about the error.

InvalidStateTransition

An update attempted to change the job execution to a state that is not valid because of the job execution's current state. For example, an attempt to change a request in state SUCCEEDED to state IN_PROGRESS. In this case, the body of the error message also contains the `executionState` field.

ResourceNotFound

The `JobExecution` specified by the request topic doesn't exist.

VersionMismatch

The expected version specified in the request doesn't match the version of the job execution in the AWS IoT Jobs service. In this case, the body of the error message also contains the `executionState` field.

InternalError

There was an internal error during the processing of the request.

RequestThrottled

The request was throttled.

TerminalStateReached

Occurs when a command to describe a job is performed on a job that is in a terminal state.

`message`

An error message string.

`clientToken`

An arbitrary string used to correlate a request with its reply.

`timestamp`

The time, in seconds since the epoch.

`executionState`

A `JobExecutionState` object. This field is included only when the `code` field has the value `InvalidStateTransition` or `VersionMismatch`. This makes it unnecessary in these cases to perform a separate `DescribeJobExecution` request to obtain the current job execution status data.

The following lists the Jobs API operations and data types.

- Jobs management and control API and data types
- Jobs device MQTT and HTTPS API operations and data types

Jobs management and control API and data types

The following commands are available for Job management and control in the CLI and over the HTTPS protocol.

- Job management and control data types
- Job management and control API operations

To determine the *endpoint-url* parameter for your CLI commands, run this command.

```
aws iot describe-endpoint --endpoint-type=iot:Jobs
```

This command returns the following output.

```
{
"endpointAddress": "account-specific-prefix.jobs.iot.aws-region.amazonaws.com"
}
```

> **Note**
>
> The Jobs endpoint doesn't support ALPN `x-amzn-http-ca`.
> If you're using dual-stack endpoints (IPv6 and IPv6), use the `iot:Data-ATS` endpoint. The `iot:Jobs` endpoint supports only IPv4.

Job management and control data types

The following data types are used by management and control applications to communicate with AWS IoT Jobs.

Job

The `Job` object contains details about a job. The following example shows the syntax:

```
{
    "jobArn": "string",
    "jobId": "string",
    "status": "IN_PROGRESS|CANCELED|SUCCEEDED",
    "forceCanceled": boolean,
    "targetSelection": "CONTINUOUS|SNAPSHOT",
    "comment": "string",
    "targets": ["string"],
    "description": "string",
    "createdAt": timestamp,
    "lastUpdatedAt": timestamp,
    "completedAt": timestamp,
    "jobProcessDetails": {
        "processingTargets": ["string"],
        "numberOfCanceledThings": long,
        "numberOfSucceededThings": long,
        "numberOfFailedThings": long,
        "numberOfRejectedThings": long,
        "numberOfQueuedThings": long,
        "numberOfInProgressThings": long,
        "numberOfRemovedThings": long,
        "numberOfTimedOutThings": long
    },
    "presignedUrlConfig": {
        "expiresInSec": number,
        "roleArn": "string"
```

```
    },
    "jobExecutionsRolloutConfig": {
        "exponentialRate": {
           "baseRatePerMinute": integer,
           "incrementFactor": integer,
           "rateIncreaseCriteria": {
              "numberOfNotifiedThings": integer, // Set one or the other
              "numberOfSucceededThings": integer // of these two values.
           },
           "maximumPerMinute": integer
      }
    },
    "abortConfig": {
       "criteriaList": [
          {
             "action": "string",
             "failureType": "string",
             "minNumberOfExecutedThings": integer,
             "thresholdPercentage": integer
          }
       ]
    },
    "SchedulingConfig": {
      "startTime": string
      "endTime": string
      "timeZone": string


      "endTimeBehavior": string

   },
    "timeoutConfig": {
        "inProgressTimeoutInMinutes": long
    }
}
```

For more information, see Job or job.

JobSummary

The `JobSummary` object contains a job summary. The following example shows the syntax:

```
{
    "jobArn": "string",
```

```
    "jobId": "string",
    "status": "IN_PROGRESS|CANCELED|SUCCEEDED|SCHEDULED",
    "targetSelection": "CONTINUOUS|SNAPSHOT",
    "thingGroupId": "string",
    "createdAt": timestamp,
    "lastUpdatedAt": timestamp,
    "completedAt": timestamp
}
```

For more information, see `JobSummary` or `job-summary`.

JobExecution

The `JobExecution` object represents the execution of a job on a device. The following example shows the syntax:

> **Note**
>
> When you use the control plane API operations, the `JobExecution` data type doesn't contain a `JobDocument` field. To obtain this information, you can use the `GetJobDocument` API operation or the `get-job-document` CLI command.

```
{
    "approximateSecondsBeforeTimedOut": 50,
    "executionNumber": 1234567890,
    "forceCanceled": true|false,
    "jobId": "string",
    "lastUpdatedAt": timestamp,
    "queuedAt": timestamp,
    "startedAt": timestamp,
    "status": "QUEUED|IN_PROGRESS|FAILED|SUCCEEDED|CANCELED|TIMED_OUT|REJECTED|
REMOVED",
    "forceCanceled": boolean,
    "statusDetails": {
        "detailsMap": {
            "string": "string" ...
        },
        "status": "string"
    },
    "thingArn": "string",
    "versionNumber": 123
```

```
}
```

For more information, see `JobExecution` or `job-execution`.

JobExecutionSummary

The `JobExecutionSummary` object contains job execution summary information. The following example shows the syntax:

```
{
    "executionNumber": 1234567890,
    "queuedAt": timestamp,
    "lastUpdatedAt": timestamp,
    "startedAt": timestamp,
    "status": "QUEUED|IN_PROGRESS|FAILED|SUCCEEDED|CANCELED|TIMED_OUT|REJECTED|REMOVED"
}
```

For more information, see `JobExecutionSummary` or `job-execution-summary`.

JobExecutionSummaryForJob

The `JobExecutionSummaryForJob` object contains a summary of information about job executions for a specific job. The following example shows the syntax:

```
{
    "executionSummaries": [
        {
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyThing",
            "jobExecutionSummary": {
                "status": "IN_PROGRESS",
                "lastUpdatedAt": 1549395301.389,
                "queuedAt": 1541526002.609,
                "executionNumber": 1
            }
        },
        ...
    ]
}
```

For more information, see `JobExecutionSummaryForJob` or `job-execution-summary-for-job`.

JobExecutionSummaryForThing

The `JobExecutionSummaryForThing` object contains a summary of information about a job execution on a specific thing. FThe following example shows the syntax:

```
{
    "executionSummaries": [
        {
            "jobExecutionSummary": {
                "status": "IN_PROGRESS",
                "lastUpdatedAt": 1549395301.389,
                "queuedAt": 1541526002.609,
                "executionNumber": 1
            },
            "jobId": "MyThingJob"
        },
        ...
    ]
}
```

For more information, see [`JobExecutionSummaryForThing`] or [`job-execution-summary-for-thing`].

Job management and control API operations

Use the following API operations or CLI commands:

AssociateTargetsWithJob

Associates a group with a continuous job. The following criteria must be met:

- The job must have been created with the `targetSelection` field set to `CONTINUOUS`.
- The job status must currently be `IN_PROGRESS`.
- The total number of targets associated with a job must not exceed 100.

HTTPS request

```
POST /jobs/jobId/targets

{
"targets": [ "string" ],
```

```
"comment": "string"
}
```

For more information, see AssociateTargetsWithJob.

CLI syntax

```
aws iot  associate-targets-with-job \
--targets <value> \
--job-id <value> \
[--comment <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"targets": [
"string"
],
"jobId": "string",
"comment": "string"
}
```

For more information, see associate-targets-with-job.

CancelJob

Cancels a job.

HTTPS request

```
PUT /jobs/jobId/cancel

{
"force": boolean,
"comment": "string",
"reasonCode": "string"
}
```

For more information, see CancelJob.

CLI syntax

```
aws iot cancel-job \
    --job-id <value> \
    [--force <value>]  \
    [--comment <value>]  \
    [--reasonCode <value>]  \
    [--cli-input-json <value>] \
    [--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
    "jobId": "string",
    "force": boolean,
    "comment": "string"
}
```

For more information, see cancel-job.

CancelJobExecution

Cancels a job execution on a device.

HTTPS request

```
PUT /things/thingName/jobs/jobId/cancel

{
"force": boolean,
"expectedVersion": "string",
"statusDetails": {
    "string": "string"
    ...
}
}
```

For more information, see CancelJobExecution.

CLI syntax

```
aws iot cancel-job-execution \
```

```
--job-id <value> \
--thing-name <value> \
[--force | --no-force] \
[--expected-version <value>] \
[--status-details <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"thingName": "string",
"force": boolean,
"expectedVersion": long,
"statusDetails": {
"string": "string"
}
}
```

For more information, see `cancel-job-execution`.

CreateJob

Creates a job. You can provide the job document as a link to a file in an Amazon S3 bucket (`documentSource` parameter), or in the body of the request (`document` parameter).

A job can be made *continuous* by setting the optional `targetSelection` parameter to `CONTINUOUS` (the default is `SNAPSHOT`). A continuous job can be used to onboard or upgrade devices as they are added to a group because it continues to run and is launched on newly added things. This can occur even after the things in the group at the time the job was created have completed the job.

A job can have an optional TimeoutConfig, which sets the value of the in-progress timer. The in-progress timer can't be updated and applies to all executions of the job.

The following validations are performed on arguments to the `CreateJob` API:

- The `targets` argument must be a list of valid thing or thing group ARNs. All things and thing groups must be in your AWS account.

- The `documentSource` argument must be a valid Amazon S3 URL to a job document. Amazon S3 URLs are in the form: `https://s3.amazonaws.com/bucketName/objectName`.
- The document stored in the URL specified by the `documentSource` argument must be a UTF-8 encoded JSON document.
- The size of a job document is limited to 32 KB due to the limit on the size of an MQTT message (128 KB) and encryption.
- The `jobId` must be unique in your AWS account.

HTTPS request

```
PUT /jobs/jobId

{
"targets": [ "string" ],
"document": "string",
"documentSource": "string",
"description": "string",
"jobTemplateArn": "string",
"presignedUrlConfigData": {
    "roleArn": "string",
    "expiresInSec": "integer"
},
"targetSelection": "CONTINUOUS|SNAPSHOT",
"jobExecutionsRolloutConfig": {
    "exponentialRate": {
       "baseRatePerMinute": integer,
       "incrementFactor": integer,
       "rateIncreaseCriteria": {
          "numberOfNotifiedThings": integer, // Set one or the other
          "numberOfSucceededThings": integer // of these two values.
       },
       "maximumPerMinute": integer
  }
},
"abortConfig": {
   "criteriaList": [
      {
         "action": "string",
         "failureType": "string",
         "minNumberOfExecutedThings": integer,
         "thresholdPercentage": integer
```

```
      }
   ]
},
"SchedulingConfig": {
    "startTime": string
    "endTime": string
    "timeZone": string


    "endTimeBehavior": string

   }
"timeoutConfig": {
  "inProgressTimeoutInMinutes": long
}
}
```

For more information, see `CreateJob`.

CLI syntax

```
aws iot create-job \
    --job-id <value> \
    --targets <value> \
    [--document-source <value>] \
    [--document <value>] \
    [--description <value>] \
    [--job-template-arn <value>] \
    [--presigned-url-config <value>] \
    [--target-selection <value>] \
    [--job-executions-rollout-config <value>] \
    [--abort-config <value>] \
    [--timeout-config <value>] \
    [--document-parameters <value>]  \
    [--cli-input-json <value>] \
    [--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
    "jobId": "string",
    "targets": [ "string" ],
    "documentSource": "string",
```

```
    "document": "string",
    "description": "string",
    "jobTemplateArn": "string",
    "presignedUrlConfig": {
        "roleArn": "string",
        "expiresInSec": long
     },
    "targetSelection": "string",
    "jobExecutionsRolloutConfig": {
          "exponentialRate": {
              "baseRatePerMinute": integer,
              "incrementFactor": integer,
              "rateIncreaseCriteria": {
                 "numberOfNotifiedThings": integer, // Set one or the other
                 "numberOfSucceededThings": integer // of these two values.
              },
      "maximumPerMinute": integer
      }
    },
    "abortConfig": {
    "criteriaList": [
        {
          "action": "string",
          "failureType": "string",
          "minNumberOfExecutedThings": integer,
          "thresholdPercentage": integer
         }
      ]
    },
    "timeoutConfig": {
          "inProgressTimeoutInMinutes": long
    },
    "documentParameters": {
    "string": "string"
    }
}
```

For more information, see `create-job`.

DeleteJob

Deletes a job and its related job executions.

Deleting a job can take time, depending on the number of job executions created for the job and various other factors. While the job is being deleted, the status of the job is shown as "DELETION_IN_PROGRESS". Attempting to delete or cancel a job whose status is already "DELETION_IN_PROGRESS" results in an error.

HTTPS request

```
DELETE /jobs/jobId?force=force
```

For more information, see DeleteJob.

CLI syntax

```
aws iot  delete-job \
--job-id <value> \
[--force | --no-force]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"force": boolean
}
```

For more information, see delete-job.

DeleteJobExecution

Deletes a job execution.

HTTPS request

```
DELETE /things/thingName/jobs/jobId/executionNumber/executionNumber?force=force
```

For more information, see DeleteJobExecution.

CLI syntax

```
aws iot  delete-job-execution \
```

```
--job-id <value> \
--thing-name <value> \
--execution-number <value> \
[--force | --no-force]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"thingName": "string",
"executionNumber": long,
"force": boolean
}
```

For more information, see `delete-job-execution`.

DescribeJob

Gets the details of the job execution.

HTTPS request

```
GET /jobs/jobId
```

For more information, see `DescribeJob`.

CLI syntax

```
aws iot describe-job \
--job-id <value>  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string"
}
```

For more information, see describe-job.

DescribeJobExecution

Gets details of a job execution. The job's execution status must be SUCCEEDED or FAILED.

HTTPS request

```
GET /things/thingName/jobs/jobId?executionNumber=executionNumber
```

For more information, see DescribeJobExecution.

CLI syntax

```
aws iot  describe-job-execution \
--job-id <value> \
--thing-name <value> \
[--execution-number <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"thingName": "string",
"executionNumber": long
}
```

For more information, see describe-job-execution.

GetJobDocument

Gets the job document for a job.

> **Note**
>
> Placeholder URLs are not replaced with presigned Amazon S3 URLs in the document returned. Presigned URLs are generated only when the AWS IoT Jobs service receives a request over MQTT.

HTTPS request

```
GET /jobs/jobId/job-document
```

For more information, see GetJobDocument.

CLI syntax

```
aws iot get-job-document \
--job-id <value>  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string"
}
```

For more information, see get-job-document.

ListJobExecutionsForJob

Gets a list of job executions for a job.

HTTPS request

```
GET /jobs/jobId/things?status=status&maxResults=maxResults&nextToken=nextToken
```

For more information, see ListJobExecutionsForJob.

CLI syntax

```
aws iot  list-job-executions-for-job \
--job-id <value> \
[--status <value>] \
[--max-results <value>] \
[--next-token <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"status": "string",
"maxResults": "integer",
"nextToken": "string"
}
```

For more information, see `list-job-executions-for-job`.

ListJobExecutionsForThing

Gets a list of job executions for a thing.

HTTPS request

```
GET /things/thingName/jobs?status=status&maxResults=maxResults&nextToken=nextToken
```

For more information, see `ListJobExecutionsForThing`.

CLI syntax

```
aws iot list-job-executions-for-thing \
--thing-name <value> \
[--status <value>] \
[--max-results <value>] \
[--next-token <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"thingName": "string",
"status": "string",
"maxResults": "integer",
"nextToken": "string"
}
```

For more information, see `list-job-executions-for-thing`.

ListJobs

Gets a list of jobs in your AWS account.

HTTPS request

```
GET /jobs?
status=status&targetSelection=targetSelection&thingGroupName=thingGroupName&thingGroupId=thi
```

For more information, see ListJobs.

CLI syntax

```
aws iot list-jobs \
[--status <value>] \
[--target-selection <value>] \
[--max-results <value>] \
[--next-token <value>] \
[--thing-group-name <value>] \
[--thing-group-id <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"status": "string",
"targetSelection": "string",
"maxResults": "integer",
"nextToken": "string",
"thingGroupName": "string",
"thingGroupId": "string"
}
```

For more information, see list-jobs.

UpdateJob

Updates supported fields of the specified job. Updated values for `timeoutConfig` take effect for only newly in-progress launches. Currently, in-progress launches continue to launch with the previous timeout configuration.

HTTPS request

```
PATCH /jobs/jobId
{
"description": "string",
"presignedUrlConfig": {
  "expiresInSec": number,
  "roleArn": "string"
},
"jobExecutionsRolloutConfig": {
  "exponentialRate": {
     "baseRatePerMinute": number,
     "incrementFactor": number,
     "rateIncreaseCriteria": {
        "numberOfNotifiedThings": number,
        "numberOfSucceededThings": number
     },
  "maximumPerMinute": number
  },
"abortConfig": {
  "criteriaList": [
     {
        "action": "string",
        "failureType": "string",
        "minNumberOfExecutedThings": number,
        "thresholdPercentage": number
     }
  ]
},
"timeoutConfig": {
  "inProgressTimeoutInMinutes": number
}
}
```

For more information, see UpdateJob.

CLI syntax

```
aws iot  update-job \
--job-id <value> \
[--description <value>] \
[--presigned-url-config <value>] \
[--job-executions-rollout-config <value>] \
[--abort-config <value>] \
```

```
[--timeout-config <value>] \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"description": "string",
"presignedUrlConfig": {
  "expiresInSec": number,
  "roleArn": "string"
},
"jobExecutionsRolloutConfig": {
  "exponentialRate": {
     "baseRatePerMinute": number,
     "incrementFactor": number,
     "rateIncreaseCriteria": {
        "numberOfNotifiedThings": number,
        "numberOfSucceededThings": number
     }
  },
  "maximumPerMinute": number
},
"abortConfig": {
  "criteriaList": [
     {
        "action": "string",
        "failureType": "string",
        "minNumberOfExecutedThings": number,
        "thresholdPercentage": number
     }
  ]
},
"timeoutConfig": {
  "inProgressTimeoutInMinutes": number
}
}
```

For more information, see `update-job`.

Jobs device MQTT and HTTPS API operations and data types

The following commands are available over the MQTT and HTTPS protocols. Use these API operations on the data plane for devices executing the jobs.

Jobs device MQTT and HTTPS data types

The following data types are used to communicate with the AWS IoT Jobs service over the MQTT and HTTPS protocols.

JobExecution

The `JobExecution` object represents the execution of a job on a device. The following example shows the syntax:

> **Note**
>
> When you use the MQTT and HTTP data plane API operations, the `JobExecution` data type contains a `JobDocument` field. Your devices can use this information to retrieve the job document from a job execution.

```
{
    "jobId" : "string",
    "thingName" : "string",
    "jobDocument" : "string",
    "status": "QUEUED|IN_PROGRESS|FAILED|SUCCEEDED|CANCELED|TIMED_OUT|REJECTED|
REMOVED",
    "statusDetails": {
        "string": "string"
    },
    "queuedAt" : "timestamp",
    "startedAt" : "timestamp",
    "lastUpdatedAt" : "timestamp",
    "versionNumber" : "number",
    "executionNumber": long
}
```

For more information, see `JobExecution` or `job-execution`.

JobExecutionState

The `JobExecutionState` contains information about the state of a job execution. The following example shows the syntax:

```
{
    "status": "QUEUED|IN_PROGRESS|FAILED|SUCCEEDED|CANCELED|TIMED_OUT|REJECTED|
REMOVED",
    "statusDetails": {
        "string": "string"
        ...
    }
    "versionNumber": "number"
}
```

For more information, see `JobExecutionState` or `job-execution-state`.

JobExecutionSummary

Contains a subset of information about a job execution. The following example shows the syntax:

```
{
    "jobId": "string",
    "queuedAt": timestamp,
    "startedAt": timestamp,
    "lastUpdatedAt": timestamp,
    "versionNumber": "number",
    "executionNumber": long
}
```

For more information, see `JobExecutionSummary` or `job-execution-summary`.

Learn more about the MQTT and HTTPS API operations in the following sections:

- Jobs device MQTT API operations
- Jobs device HTTP API

Jobs device MQTT API operations

You can issue jobs device commands by publishing MQTT messages to the Reserved topics used for Jobs commands.

Your device-side client must be subscribed to the response message topics of these commands. If you use the AWS IoT Device Client, your device will automatically subscribe to the response topics. This means that the message broker will publish response message topics to the client that published the command message, whether or not your client has subscribed to the response message topics. These response messages don't pass through the message broker and can't be subscribed to by other clients or rules.

When subscribing to the job and `jobExecution` event topics for your fleet-monitoring solution, first enable job and job execution events to receive any events on the cloud side. Job progress messages that are processed through the message broker and can be used by AWS IoT rules are published as Jobs events. Because the message broker publishes response messages, even without an explicit subscription to them, your client must be configured to receive and identify the messages it receives. Your client must also confirm that the *`thingName`* in the incoming message topic applies to the client's thing name before the client acts on the message.

> **Note**
>
> Messages that AWS IoT sends in response to MQTT Jobs API command messages are charged to your account, whether or not you subscribed to them explicitly.

The following shows the MQTT API operations and their request and response syntax. All MQTT API operations have the following parameters:

clientToken

An optional client token used to correlate requests and responses. Enter an arbitrary value here and it's reflected in the response.

`timestamp`

The time in seconds since the epoch, when the message was sent.

GetPendingJobExecutions

Gets the list of all jobs that are not in a terminal state, for a specified thing.

To invoke this API, publish a message on `$aws/things/`*`thingName`*`/jobs/get`.

Request payload:

```
{ "clientToken": "string" }
```

The message broker will publish `$aws/things/thingName/jobs/get/accepted` and `$aws/things/thingName/jobs/get/rejected` even without a specific subscription to them. However, for your client to receive the messages, it must be listening for them. For more information, see the note about Jobs API messages.

Response payload:

```
{
"inProgressJobs" : [ JobExecutionSummary ... ],
"queuedJobs" : [ JobExecutionSummary ... ],
"timestamp" : 1489096425069,
"clientToken" : "client-001"
}
```

Where `inProgressJobs` and `queuedJobs` return a list of JobExecutionSummary objects that have status of `IN_PROGRESS` or `QUEUED`.

StartNextPendingJobExecution

Gets and starts the next pending job execution for a thing (status `IN_PROGRESS` or `QUEUED`).

- Any job executions with status `IN_PROGRESS` are returned first.
- Job executions are returned in the order in which they were queued. When a thing is added or removed from the target group for your job, confirm the rollout order of any new job executions compared to existing job executions.
- If the next pending job execution is `QUEUED`, its state changes to `IN_PROGRESS` and the job execution's status details are set as specified.
- If the next pending job execution is already `IN_PROGRESS`, its status details aren't changed.
- If no job executions are pending, the response doesn't include the `execution` field.
- Optionally, you can create a step timer by setting a value for the `stepTimeoutInMinutes` property. If you don't update the value of this property by running `UpdateJobExecution`, the job execution times out when the step timer expires.

To invoke this API, publish a message on `$aws/things/thingName/jobs/start-next`.

Request payload:

```
{
"statusDetails": {
    "string": "job-execution-state"
    ...
},
"stepTimeoutInMinutes": long,
"clientToken": "string"
}
```

`statusDetails`

A collection of name-value pairs that describe the status of the job execution. If not specified, the `statusDetails` are unchanged.

`stepTimeOutInMinutes`

Specifies the amount of time this device has to finish execution of this job. If the job execution status isn't set to a terminal state before this timer expires, or before the timer is reset, (by calling `UpdateJobExecution`, setting the status to `IN_PROGRESS` and specifying a new timeout value in field `stepTimeoutInMinutes`) the job execution status is set to `TIMED_OUT`. Setting this timeout has no effect on that job execution timeout that might have been specified when the job was created (`CreateJob` using the `timeoutConfig` field).

Valid values for this parameter range from 1 to 10080 (1 minute to 7 days). A value of -1 is also valid and will cancel the current step timer (created by an earlier use of UpdateJobExecutionRequest).

The message broker will publish `$aws/things/thingName/jobs/start-next/accepted` and `$aws/things/thingName/jobs/start-next/rejected` even without a specific subscription to them. However, for your client to receive the messages, it must be listening for them. For more information, see [the note about Jobs API messages](#).

Response payload:

```
{
"execution" : JobExecutionData,
"timestamp" : timestamp,
"clientToken" : "string"
```

```
}
```

Where `execution` is a [JobExecution](#) object. For example:

```
{
"execution" : {
    "jobId" : "022",
    "thingName" : "MyThing",
    "jobDocument" : "< contents of job document >",
    "status" : "IN_PROGRESS",
    "queuedAt" : 1489096123309,
    "lastUpdatedAt" : 1489096123309,
    "versionNumber" : 1,
    "executionNumber" : 1234567890
},
"clientToken" : "client-1",
"timestamp" : 1489088524284,
}
```

DescribeJobExecution

Gets detailed information about a job execution.

You can set the `jobId` to `$next` to return the next pending job execution for a thing (with a status of `IN_PROGRESS` or `QUEUED`).

To invoke this API, publish a message on `$aws/things/thingName/jobs/jobId/get`.

Request payload:

```
{
"jobId" : "022",
"thingName" : "MyThing",
"executionNumber": long,
"includeJobDocument": boolean,
"clientToken": "string"
}
```

`thingName`

The name of the thing associated with the device.

`jobId`

The unique identifier assigned to this job when it was created.

Or use `$next` to return the next pending job execution for a thing (with a status of `IN_PROGRESS` or `QUEUED`). In this case, any job executions with status `IN_PROGRESS` are returned first. Job executions are returned in the order in which they were created.

`executionNumber`

(Optional) A number that identifies a job execution on a device. If not specified, the latest job execution is returned.

`includeJobDocument`

(Optional) Unless set to `false`, the response contains the job document. The default is `true`.

The message broker will publish `$aws/things/thingName/jobs/jobId/get/accepted` and `$aws/things/thingName/jobs/jobId/get/rejected` even without a specific subscription to them. However, for your client to receive the messages, it must be listening for them. For more information, see the note about Jobs API messages.

Response payload:

```
{
"execution" : JobExecutionData,
"timestamp": "timestamp",
"clientToken": "string"
}
```

Where `execution` is a JobExecution object.

UpdateJobExecution

Updates the status of a job execution. You can optionally create a step timer by setting a value for the `stepTimeoutInMinutes` property. If you don't update the value of this property by running `UpdateJobExecution` again, the job execution times out when the step timer expires.

To invoke this API, publish a message on `$aws/things/thingName/jobs/jobId/update`.

Request payload:

```
{
```

```
"status": "job-execution-state",
"statusDetails": {
    "string": "string"
    ...
},
"expectedVersion": "number",
"executionNumber": long,
"includeJobExecutionState": boolean,
"includeJobDocument": boolean,
"stepTimeoutInMinutes": long,
"clientToken": "string"
}
```

`status`

The new status for the job execution (`IN_PROGRESS`, `FAILED`, `SUCCEEDED`, or `REJECTED`). This must be specified on every update.

`statusDetails`

A collection of name-value pairs that describe the status of the job execution. If not specified, the `statusDetails` are unchanged.

`expectedVersion`

The expected current version of the job execution. Each time you update the job execution, its version is incremented. If the version of the job execution stored in the AWS IoT Jobs service doesn't match, the update is rejected with a `VersionMismatch` error. An [ErrorResponse] that contains the current job execution status data is also returned. (This makes it unnecessary to perform a separate `DescribeJobExecution` request to obtain the job execution status data.)

`executionNumber`

(Optional) A number that identifies a job execution on a device. If not specified, the latest job execution is used.

`includeJobExecutionState`

(Optional) When included and set to `true`, the response contains the `JobExecutionState` field. The default is `false`.

`includeJobDocument`

(Optional) When included and set to `true`, the response contains the `JobDocument`. The default is `false`.

`stepTimeoutInMinutes`

Specifies the amount of time this device has to finish execution of this job. If the job execution status is not set to a terminal state before this timer expires, or before the timer is reset, the job execution status is set to `TIMED_OUT`. Setting or resetting this timeout has no effect on the job execution timeout that might have been specified when the job was created.

The message broker will publish `$aws/things/thingName/jobs/jobId/update/accepted` and `$aws/things/thingName/jobs/jobId/update/rejected` even without a specific subscription to them. However, for your client to receive the messages, it must be listening for them. For more information, see the note about Jobs API messages.

Response payload:

```
{
"executionState": JobExecutionState,
"jobDocument": "string",
"timestamp": timestamp,
"clientToken": "string"
}
```

`executionState`

A JobExecutionState object.

`jobDocument`

A job document object.

Note

In MQTT responses, the `jobDocument` field is a JSON object. In HTTP responses, it's a string representation of the JSON object.

`timestamp`

The time in seconds since the epoch, when the message was sent.

`clientToken`

A client token used to correlate requests and responses.

When you use the MQTT protocol, you can also perform the following updates:

JobExecutionsChanged

Sent whenever a job execution is added to or removed from the list of pending job executions for a thing.

Use the topic:

`$aws/things/thingName/jobs/notify`

Message payload:

```
{
"jobs" : {
    "JobExecutionState": [ JobExecutionSummary ... ]
         },
    "timestamp": timestamp
}
```

NextJobExecutionChanged

Sent whenever there is a change to which job execution is next on the list of pending job executions for a thing, as defined for `DescribeJobExecution` with `jobId` `$next`. This message is not sent when the next job's execution details change, only when the next job that would be returned by `DescribeJobExecution` with `jobId` `$next` has changed. Consider job executions J1 and J2 with a status of `QUEUED`. J1 is next on the list of pending job executions. If the status of J2 is changed to `IN_PROGRESS` while the state of J1 remains unchanged, then this notification is sent and contains details of J2.

Use the topic:

`$aws/things/thingName/jobs/notify-next`

Message payload:

```
{
"execution" : JobExecution,
"timestamp": timestamp,
}
```

Jobs device HTTP API

Devices can communicate with AWS IoT Jobs using HTTP Signature Version 4 on port 443. This is the method used by the AWS SDKs and CLI. For more information about those tools, see AWS CLI Command Reference:iot-jobs-data or AWS SDKs and Tools.

The following commands are available for devices executing the jobs. For information about using API operations with the MQTT protocol, see Jobs device MQTT API operations.

GetPendingJobExecutions

Gets the list of all jobs that aren't in a terminal state, for a specified thing.

HTTPS request

```
GET /things/thingName/jobs
```

Response:

```
{
"inProgressJobs" : [ JobExecutionSummary ... ],
"queuedJobs" : [ JobExecutionSummary ... ]
}
```

For more information, see `GetPendingJobExecutions`.

CLI syntax

```
aws iot-jobs-data get-pending-job-executions \
--thing-name <value>  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"thingName": "string"
}
```

For more information, see `get-pending-job-executions`.

StartNextPendingJobExecution

Gets and starts the next pending job execution for a thing (with a status of `IN_PROGRESS` or `QUEUED`).

- Any job executions with status `IN_PROGRESS` are returned first.
- Job executions are returned in the order in which they were created.
- If the next pending job execution is `QUEUED`, its status changes to `IN_PROGRESS` and the job execution's status details are set as specified.
- If the next pending job execution is already `IN_PROGRESS`, its status details don't change.
- If no job executions are pending, the response doesn't include the `execution` field.
- Optionally, you can create a step timer by setting a value for the `stepTimeoutInMinutes` property. If you don't update the value of this property by running `UpdateJobExecution`, the job execution times out when the step timer expires.

HTTPS request

The following example shows the request syntax:

```
PUT /things/thingName/jobs/$next
{
"statusDetails": {
    "string": "string"
    ...
},
"stepTimeoutInMinutes": long
}
```

For more information, see [StartNextPendingJobExecution]().

CLI syntax

Synopsis:

```
aws iot-jobs-data  start-next-pending-job-execution \
--thing-name <value> \
{--step-timeout-in-minutes <value>] \
[--status-details <value>]  \
[--cli-input-json <value>] \
```

```
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"thingName": "string",
"statusDetails": {
"string": "string"
},
"stepTimeoutInMinutes": long
}
```

For more information, see `start-next-pending-job-execution`.

DescribeJobExecution

Gets detailed information about a job execution.

You can set the `jobId` to `$next` to return the next pending job execution for a thing. The job's execution status must be `QUEUED` or `IN_PROGRESS`.

HTTPS request

Request:

```
GET /things/thingName/jobs/jobId?
executionNumber=executionNumber&includeJobDocument=includeJobDocument
```

Response:

```
{
"execution" : JobExecution,
}
```

For more information, see `DescribeJobExecution`.

CLI syntax

Synopsis:

```
aws iot-jobs-data  describe-job-execution \
```

```
--job-id <value> \
--thing-name <value> \
[--include-job-document | --no-include-job-document] \
[--execution-number <value>]  \
[--cli-input-json <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"thingName": "string",
"includeJobDocument": boolean,
"executionNumber": long
}
```

For more information, see describe-job-execution.

UpdateJobExecution

Updates the status of a job execution. Optionally, you can create a step timer by setting a value for the `stepTimeoutInMinutes` property. If you don't update the value of this property by running `UpdateJobExecution` again, the job execution times out when the step timer expires.

HTTPS request

Request:

```
POST /things/thingName/jobs/jobId
{
"status": "job-execution-state",
"statusDetails": {
    "string": "string"
    ...
},
"expectedVersion": "number",
"includeJobExecutionState": boolean,
"includeJobDocument": boolean,
"stepTimeoutInMinutes": long,
"executionNumber": long
}
```

For more information, see [UpdateJobExecution].

CLI syntax

Synopsis:

```
aws iot-jobs-data  update-job-execution \
--job-id <value> \
--thing-name <value> \
--status <value> \
[--status-details <value>] \
[--expected-version <value>] \
[--include-job-execution-state | --no-include-job-execution-state] \
[--include-job-document | --no-include-job-document] \
[--execution-number <value>]  \
[--cli-input-json <value>] \
[--step-timeout-in-minutes <value>] \
[--generate-cli-skeleton]
```

`cli-input-json` format:

```
{
"jobId": "string",
"thingName": "string",
"status": "string",
"statusDetails": {
"string": "string"
},
"stepTimeoutInMinutes": number,
"expectedVersion": long,
"includeJobExecutionState": boolean,
"includeJobDocument": boolean,
"executionNumber": long
}
```

For more information, see [update-job-execution].

Securing users and devices with AWS IoT Jobs

To authorize users to use AWS IoT Jobs with their devices, you must grant them permissions by using IAM policies. The devices must then be authorized by using AWS IoT Core policies to connect securely to AWS IoT, receive job executions, and update the execution status.

Required policy type for AWS IoT Jobs

The following table shows the different types of policies that you must use for authorization. For more information about the required policy to use, see Authorization.

Required policy type

Use case	Protocol	Authentication	Control plane/data plane	Identity type	Required policy type
Authorize an administrator, operator, or Cloud Service to work securely with Jobs	HTTPS	AWS Signature Version 4 authentication (port 443)	Both control plane and data plane	Amazon Cognito Identity, IAM, or federated user	IAM policy
Authorize your IoT device to work securely with Jobs	MQTT/ HTTPS	TCP or TLS mutual authentication (port 8883 or 443)	Data plane	X.509 certificates	AWS IoT Core policy

To authorize AWS IoT Jobs operations that can be performed both on the control plane and data plane, you must use IAM policies. The identities must have been authenticated with AWS IoT to perform these operations, which must be Amazon Cognito identities or IAM users, groups, and roles. For more information about authentication, see Authentication.

The devices must now be authorized on the data plane by using AWS IoT Core policies to connect securely to the device gateway. The device gateway enables devices to securely communicate with AWS IoT, receive job executions, and update the job execution status. Device communication is secured by using secure MQTT or HTTPS publish communication protocols. These protocols use X.509 client certificates that are provided by AWS IoT to authenticate the device connections.

The following shows how you authorize your users, cloud services, and devices to use AWS IoT Jobs. For information about control plane and data plane API operations, see AWS IoT jobs API operations.

Topics

- Authorizing users and cloud services to use AWS IoT Jobs
- Authorizing your devices to securely use AWS IoT Jobs on the data plane

Authorizing users and cloud services to use AWS IoT Jobs

To authorize your users and cloud services, you must use IAM policies on both the control plane and data plane. The policies must be used with HTTPS protocol and must use AWS Signature Version 4 authentication (port 443) to authenticate users.

> **Note**
>
> AWS IoT Core policies must not be used on the control plane. Only IAM policies are used for authorizing users or Cloud Services. For more information about using the required policy type, see Required policy type for AWS IoT Jobs.

IAM policies are JSON documents that contain policy statements. Policy statements use *Effect*, *Action*, and *Resource* elements to specify resources, allowed or denied actions, and conditions under which actions are allowed or denied. For more information, see IAM JSON Policy Elements Reference in the *IAM user Guide*.

> **Warning**
>
> We recommend that you don't use wildcard permissions, such as `"Action": ["iot:*"]` in your IAM policies or AWS IoT Core policies. Using wildcard permissions is not a recommended security best practice. For more information, see AWS IoT policy overly permissive.

IAM policies on the control plane

On the control plane, IAM policies use the `iot:` prefix with the action to authorize the corresponding jobs API operation. For example, the `iot:CreateJob` policy action grants the user permission to use the [CreateJob](#) API.

Policy actions

The following table shows a list of IAM policy actions and permissions to use the API actions. For information about resource types, see [Resource types defined by AWS IoT](#). For more information about AWS IoT actions, see [Actions defined by AWS IoT](#).

IAM policy actions on control plane

Policy action	API operation	Resource types	Description
`iot:AssociateTargetsWithJob`	[AssociateTargetsWithJob](#)	• job • thing • thinggroup	Represents the permission to associate a group with a continuous job. The `iot:AssociateTargetsWithJob` permission is checked every time a request is made to associate targets.
`iot:CancelJob`	[CancelJob](#)	job	Represents the permission to cancel a job. The `iot:CancelJob` permission is checked every time a request is made to cancel a job.
`iot:CancelJobExecution`	[CancelJobExecution](#)	• job • thing	Represents the permission to cancel a job execution. The `iot: CancelJobExecution` permission is checked every time a request is made to cancel a job execution.
`iot:CreateJob`	[CreateJob](#)	• job • thing • thinggroup	Represents the permission to create a job. The `iot: CreateJob` permission is checked every time a request is made to create a job.

Policy action	API operation	Resource types	Description
		• jobtemplate • package	
iot:CreateJobTemplate	CreateJobTemplate	• job • jobtemplate • package	Represents the permission to create a job template. The iot: CreateJobTemplate permission is checked every time a request is made to create a job template.
iot:DeleteJob	DeleteJob	job	Represents the permission to delete a job. The iot: DeleteJob permission is checked every time a request is made to delete a job.
iot:DeleteJobTemplate	DeleteJobTemplate	jobtemplate	Represents the permission to delete a job template. The iot: CreateJobTemplate permission is checked every time a request is made to delete a job template.
iot:DeleteJobExecution	DeleteJobTemplate	• job • thing	Represents the permission to delete a job execution. The iot: DeleteJobExecution permission is checked every time a request is made to delete a job execution.
iot:DescribeJob	DescribeJob	job	Represents the permission to describe a job. The iot: DescribeJob permission is checked every time a request is made to describe a job.

Policy action	API operation	Resource types	Description
iot:DescribeJobExecution	DescribeJobExecution	• job • thing	Represents the permission to describe a job execution. The iot: DescribeJobExecution permission is checked every time a request is made to describe a job execution.
iot:DescribeJobTemplate	DescribeJobTemplate	jobtemplate	Represents the permission to describe a job template. The iot: DescribeJobTemplate permission is checked every time a request is made to describe a job template.
iot:DescribeManagedJobTemplate	DescribeManagedJobTemplate	jobtemplate	Represents the permission to describe a managed job template. The iot: DescribeManagedJobTemplate permission is checked every time a request is made to describe a managed job template.
iot:GetJobDocument	GetJobDocument	job	Represents the permission to get the job document for a job. The iot:GetJobDocument permission is checked every time a request is made to get a job document.
iot:ListJobExecutionsForJob	ListJobExecutionsForJob	job	Represents the permission to list the job executions for a job. The iot:ListJobExecutionsForJob permission is checked every time a request is made to list the job executions for a job.

Policy action	API operation	Resource types	Description
iot:ListJobExecutionsForThing	[ListJobExecutionsForThing]()	thing	Represents the permission to list the job executions for a job. The `iot:ListJobExecutionsForThing` permission is checked every time a request is made to list the job executions for a thing.
iot:ListJobs	[ListJobs]()	none	Represents the permission to list the jobs. The `iot:ListJobs` permission is checked every time a request is made to list the jobs.
iot:ListJobTemplates	[ListJobTemplates]()	none	Represents the permission to list the job templates. The `iot:ListJobTemplates` permission is checked every time a request is made to list the job templates.
iot:ListManagedJobTemplates	[ListManagedJobTemplates]()	none	Represents the permission to list the managed job templates. The `iot:ListManagedJobTemplates` permission is checked every time a request is made to list the managed job templates.
iot:UpdateJob	[UpdateJob]()	job	Represents the permission to update a job. The `iot:UpdateJob` permission is checked every time a request is made to update a job.
iot:TagResource	[TagResource]()	• job • jobtemplate • thing	Grants permission to tag a specific resource.

Policy action	API operation	Resource types	Description
iot:UntagResource	[UntagResource](#)	• job • jobtemplate • thing	Grants permission to untag a specific resource.

Basic IAM policy example

The following example shows an IAM policy that allows the user permission to perform the following actions for your IoT thing and thing group.

In the example, replace:

- *region* with your AWS Region, such as `us-east-1`.
- *account-id* with your AWS account number, such as `57EXAMPLE833`.
- *thing-group-name* with the name of your IoT thing group for which you're targeting jobs, such as `FirmwareUpdateGroup`.
- *thing-name* with the name of your IoT thing for which you're targeting jobs, such as `MyIoTThing`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "iot:CreateJobTemplate",
                "iot:CreateJob"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:us-east-1:123456789012:thinggroup/thing-group-
name"
        },
        {
            "Action": [
                "iot:DescribeJob",
```

```
                "iot:CancelJob",
                "iot:DeleteJob"
            ],
            "Effect": "Allow",
     "Resource": "arn:aws:iot:us-east-1:123456789012:job/*"
        },
        {
            "Action": [
                "iot:DescribeJobExecution",
                "iot:CancelJobExecution",
                "iot:DeleteJobExecution"
            ],
            "Effect": "Allow",
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:thing/thing-123",
                "arn:aws:iot:us-east-1:123456789012:job/*"
            ]
        }
    ]
}
```

IAM policy example for IP based authorization

You can restrict *principals* from making API calls to your control plane endpoint from specific IP addresses. To specify the IP addresses that can be allowed, in the Condition element of your IAM policy, use the `aws:SourceIp` global condition key.

Using this condition key can also deny access to other AWS services from making these API calls on your behalf, such as AWS CloudFormation. To allow access to these services, use the `aws:ViaAWSService` global condition key with the aws:SourceIp key. This makes sure that the source IP address access restriction applies only to requests that are made directly by a principal. For more information, see AWS: Denies access to AWS based on the source IP.

The following example shows how to allow only a specific IP address that can make API calls to the control plane endpoint. The `aws:ViaAWSService` key is set to `true`, which allows other services to make API calls on your behalf.

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "iot:CreateJobTemplate",
                "iot:CreateJob"
            ],
            "Resource": [
                "*"
            ],
            "Condition": {
                 "IpAddress": {
                     "aws:SourceIp": "123.45.167.89"
             },
                 "Bool": {
                     "aws:ViaAWSService": "false"
        }
        }
    }]
}
```

IAM policies on the data plane

IAM policies on the data plane use the `iotjobsdata:` prefix to authorize jobs API operations that users can perform. On the data plane, you can grant a user permission to use the [`DescribeJobExecution`]() API by using the `iotjobsdata:DescribeJobExecution` policy action.

> **Warning**
>
> Using IAM policies on the data plane is not recommended when targeting AWS IoT Jobs for your devices. We recommend that you use IAM policies on the control plane for users to create and manage jobs. On the data plane, for authorizing devices to retrieve job executions and update the execution status, use [AWS IoT Core policies for HTTPS protocol]().

Basic IAM policy example

The API operations that must be authorized are usually performed by you typing CLI commands. The following shows an example of a user performing a `DescribeJobExecution` operation.

In the example, replace:

- *region* with your AWS Region, such as `us-east-1`.
- *account-id* with your AWS account number, such as `57EXAMPLE833`.
- *thing-name* with the name of your IoT thing for which you're targeting jobs, such as `myRegisteredThing`.
- *job-id* is the unique identifier for the job that's targeted using the API.

```
aws iot-jobs-data describe-job-execution \
    --endpoint-url "https://account-id.jobs.iot.region.amazonaws.com" \
    --job-id jobID --thing-name thing-name
```

The following shows a sample IAM policy that authorizes this action:

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": [
            "iotjobsdata:DescribeJobExecution"
        ],
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:thing/thing-123"
    }
}
```

IAM policy examples for IP based authorization

You can restrict *principals* from making API calls to your data plane endpoint from specific IP addresses. To specify the IP addresses that can be allowed, in the Condition element of your IAM policy, use the [`aws:SourceIp`] global condition key.

Using this condition key can also deny access to other AWS services from making these API calls on your behalf, such as AWS CloudFormation. To allow access to these services, use the [`aws:ViaAWSService`] global condition key with the `aws:SourceIp` condition key. This makes sure that the IP address access restriction only applies to requests that are directly made by the principal. For more information, see [AWS: Denies access to AWS based on the source IP].

The following example shows how to allow only a specific IP address that can make API calls to the data plane endpoint.

```
{
    "Version":"2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iotjobsdata:*"
        ],
        "Resource": [
            "*"
        ],
        "Condition": {
            "IpAddress": {
                "aws:SourceIp": "123.45.167.89"
        },
            "Bool": {
                "aws:ViaAWSService": "false"
        }
        }
    }]
}
```

The following example shows how to restrict specific IP addresses or address ranges from making API calls to the data plane endpoint.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iotjobsdata:*"
            ],
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": [
                        "123.45.167.89",
                        "192.0.2.0/24",
```

```
                            "203.0.113.0/24"
                        ]
                }
            },
                "Resource": [
                    "*"
                ]
            }
        ]
}
```

IAM policy example for both control plane and data plane

If you perform an API operation on both the control plane and data plane, your control plane policy action must use the `iot:` prefix, and your data plane policy action must use the `iotjobsdata:` prefix.

For example, the `DescribeJobExecution` API can be used in both the control plane and data plane. On the control plane, the [DescribeJobExecution](#) API is used to describe a job execution. On the data plane, the [DescribeJobExecution](#) API is used to get details of a job execution.

The following IAM policy authorizes a user permission to use the `DescribeJobExecution` API on both the control plane and data plane.

In the example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as `57EXAMPLE833`.
- *`thing-name`* with the name of your IoT thing for which you're targeting jobs, such as `MyIoTThing`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "iotjobsdata:DescribeJobExecution"
            ],
```

```
            "Effect": "Allow",
            "Resource": "arn:aws:iot:us-east-1:123456789012:thing/thing-123"
        },
        {
            "Action": [
                "iot:DescribeJobExecution",
                "iot:CancelJobExecution",
                "iot:DeleteJobExecution"
            ],
            "Effect": "Allow",
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:thing/thing-123",
    "arn:aws:iot:us-east-1:123456789012:job/*"
            ]
        }
    ]
}
```

Authorize tagging of IoT resources

For better control over jobs and job templates that you can create, modify, or use, you can attach tags to the jobs or job templates. Tags also help you discern ownership and assign and allocate costs by placing them in billing groups and attaching tags to them.

When a user wants to tag their jobs or job templates that they created by using the AWS Management Console or the AWS CLI, your IAM policy must grant the user permissions to tag them. To grant permissions, your IAM policy must use the `iot:TagResource` action.

> **Note**
>
> If your IAM policy doesn't include the `iot:TagResource` action, then any `CreateJob` or `CreateJobTemplate` with a tag will return an `AccessDeniedException` error.

When you want to tag your jobs or job templates that you created by using the AWS Management Console or the AWS CLI, your IAM policy must grant permission to tag them. To grant permissions, your IAM policy must use the `iot:TagResource` action.

For general information about tagging your resources, see Tagging your AWS IoT resources.

IAM policy example

Refer to the following IAM policy examples granting tagging permissions:

Example 1

A user that runs the following command to create a job and tag it to a specific environment.

In this example, replace:

- *region* with your AWS Region, such as us-east-1.
- *account-id* with your AWS account number, such as 57EXAMPLE833.
- *thing-name* with the name of your IoT thing for which you're targeting jobs, such as MyIoTThing.

```
aws iot create-job
    --job-id test_job
    --targets "arn:aws:iot:region:account-id:thing/thingOne"
    --document-source "https://s3.amazonaws.com/amzn-s3-demo-bucket/job-document.json"
    --description "test job description"
    --tags Key=environment,Value=beta
```

For this example, you must use the following IAM policy:

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": [
            "iot:CreateJob",
            "iot:CreateJobTemplate",
            "iot:TagResource"
        ],
        "Effect": "Allow",
        "Resource": [
           "arn:aws:iot:us-east-1:123456789012:job/*",
           "arn:aws:iot:us-east-1:123456789012:jobtemplate/*"
        ]
    }
}
```

Authorizing your devices to securely use AWS IoT Jobs on the data plane

To authorize your devices to interact securely with AWS IoT Jobs on the data plane, you must use AWS IoT Core policies. AWS IoT Core policies for jobs are JSON documents containing policy statements. These policies also use *Effect*, *Action*, and *Resource* elements, and follow a similar convention to IAM policies. For more information about the elements, see IAM JSON Policy Elements Reference in the *IAM user Guide*.

The policies can be used with both MQTT and HTTPS protocols and must use TCP or TLS mutual authentication to authenticate the devices. The following shows how to use these policies across the different communication protocols.

> **Warning**
>
> We recommend that you don't use wildcard permissions, such as `"Action": ["iot:*"]` in your IAM policies or AWS IoT Core policies. Using wildcard permissions is not a recommended security best practice. For more information, see AWS IoT policy overly permissive.

AWS IoT Core policies for MQTT protocol

AWS IoT Core policies for MQTT protocol grant you permissions to use the jobs device MQTT API actions. The MQTT API operations are used to work with MQTT topics that are reserved for jobs commands. For more information about these API operations, see Jobs device MQTT API operations.

MQTT policies use policy actions such as `iot:Connect`, `iot:Publish`, `iot:Subscribe`, and `iot:Receieve` to work with the jobs topics. These policies allow you to connect to the message broker, subscribe to the jobs MQTT topics, and send and receive MQTT messages between your devices and the cloud. For more information about these actions, see AWS IoT Core policy actions.

For information about topics for AWS IoT Jobs, see Job topics.

Basic MQTT policy example

The following example shows how you can use `iot:Publish` and `iot:Subscribe` to publish and subscribe to jobs and job executions.

In the example, replace:

- *region* with your AWS Region, such as `us-east-1`.
- *account-id* with your AWS account number, such as `57EXAMPLE833`.
- *thing-name* with the name of your IoT thing for which you're targeting jobs, such as `MyIoTThing`.

```
{
    "Version":"2012-10-17",

    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Subscribe"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/$aws/events/job/*",
    "arn:aws:iot:us-east-1:123456789012:topic/$aws/events/jobExecution/*",
    "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/thing-123/jobs/*"
            ]
        }
    ]
}
```

AWS IoT Core policies for HTTPS protocol

AWS IoT Core policies on the data plane can also use the HTTPS protocol with the TLS authentication mechanism to authorize your devices. On the data plane, policies use the `iotjobsdata:` prefix to authorize jobs API operations that your devices can perform. For example, the `iotjobsdata:DescribeJobExecution` policy action grants the user permission to use the `DescribeJobExecution` API.

> **Note**
>
> The data plane policy actions must use the `iotjobsdata:` prefix. On the control plane, the actions must use the `iot:` prefix. For an example IAM policy when both control plane

and data plane policy actions are used, see IAM policy example for both control plane and data plane.

Policy actions

The following table shows a list of AWS IoT Core policy actions and permissions for authorizing devices to use the API actions. For a list of API operations that you can perform in the data plane, see Jobs device HTTP API.

> **Note**
>
> These job execution policy actions apply only to the HTTP TLS endpoint. If you use the MQTT endpoint, you must use the MQTT policy actions defined previously.

AWS IoT Core policy actions on data plane

Policy action	API operation	Resource types	Description
`iotjobsdata:DescribeJobExecution`	DescribeJobExecution	• job • thing	Represents the permission to retrieve a job execution. The `iotjobsdata:DescribeJobExecution` permission is checked every time a request is made to retrieve a job execution.
`iotjobsdata:GetPendingJobExecutions`	GetPendingJobExecutions	thing	Represents the permission to retrieve the list of jobs that are not in a terminal status for a thing. The `iotjobsdata:GetPendingJobExecutions` permission is checked every time a request is made to retrieve the list.
`iotjobsdata:StartN`	StartNextPendingJobExecution	thing	Represents the permission to get and start the next pending job execution for a thing. That is,

Policy action	API operation	Resource types	Description
extPendingJobExecution			to update a job execution with status QUEUED to IN_PROGRESS . The iotjobsdata:StartNextPendingJobExecution permission is checked every time a request is made to start the next pending job execution.
iotjobsdata:UpdateJobExecution	UpdateJobExecution	thing	Represents the permission to update a job execution. The iotjobsdata:UpdateJobExecution permission is checked every time a request is made to update the state of a job execution.

Basic policy example

The following shows an example of an AWS IoT Core policy that grants permission to perform the actions on the data plane API operations for any resource. You can scope your policy to a specific resource, such as an IoT thing. In your example, replace:

- *region* with your AWS Region such as us-east-1.
- *account-id* with your AWS account number, such as 57EXAMPLE833.
- *thing-name* with the name of the IoT thing, such as MyIoTthing.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "iotjobsdata:GetPendingJobExecutions",
                "iotjobsdata:StartNextPendingJobExecution",
                "iotjobsdata:DescribeJobExecution",
                "iotjobsdata:UpdateJobExecution"
```

```
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:us-east-1:123456789012:thing/thing-123"
        }
    ]
}
```

An example of when you must use these policies can be when your IoT devices use an AWS IoT Core policy to access one of these API operations, such as the following example of the `DescribeJobExecution` API:

```
GET /things/thingName/jobs/jobId?
executionNumber=executionNumber&includeJobDocument=includeJobDocument&namespaceId=namespaceId
 HTTP/1.1
```

AWS IoT Jobs limits

AWS IoT Jobs has Service quotas, or limits, that correspond to the maximum number of service resources or operations for your AWS account.

Topics

- Job executions limits
- Active and concurrent job limits

Job executions limits

This section provides information about the job execution limits for AWS IoT Device Management.

> **Note**
>
> These limits are not part of the service quotas that you can find in the AWS IoT Device Management Service Quotas documentation.

To get information about the number of pending job executions, you can either use the `GetPendingJobExecutions` API, or subscribe to the MQTT reserved topics for AWS IoT Jobs and receive Job notification types.

The number of pending job executions in your account can vary depending on whether you have the scheduling configuration enabled and use a recurring maintenance window.

Maximum number of pending job executions

API/notification name	Description	Without scheduling configuration	With scheduling configuration
ListNotification	A ListNotification is published whenever an old job execution enters a terminal status, or when a new job execution is queued or changes to a non-terminal status. It can display up to 15 pending job executions that are either QUEUED or IN_PROGRESS.	10	15 (Up to 5 job executions only appears in the ListNotification during a maintenace window).
GetPendingJobExecutions	When you invoke the GetPendingJobExecutions API, it returns a list of job executions that have not yet started, and can be started after the API call. The API can return up to a maximum of 10 pending job executions. • Out of the 10 pending job executions, executions that are IN_PROGRESS will be filtered from the result. • Out of the 10 pending job executions, if their jobs are in SCHEDULED status, they will be filtered from the result.	10	15

Active and concurrent job limits

This section will help you learn more about active and concurrent jobs and the limits that apply to them.

Active jobs and active job limit

When you create a job by using the AWS IoT console or the `CreateJob` API, the job status changes to `IN_PROGRESS`. All in-progress jobs are *active jobs* and count towards the active jobs limit. This includes jobs that are either rolling out new job executions, or jobs that are waiting for devices to complete their job executions. This limit applies to both continuous and snapshot jobs.

Concurrent jobs and job concurrency limit

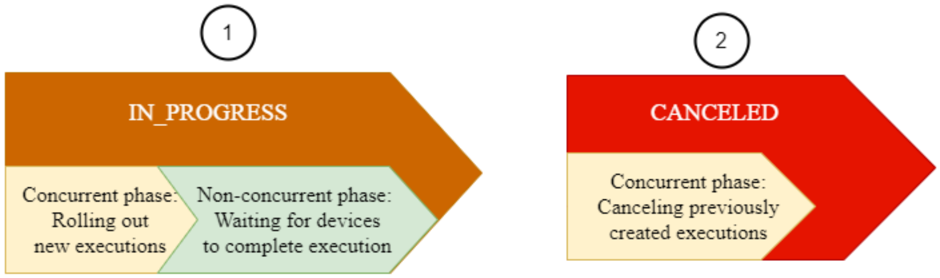
In-progress jobs that are either rolling out new job executions, or jobs that are canceling previously created job executions are *concurrent jobs* and count towards the job concurrency limit. AWS IoT Jobs can roll out and cancel job executions swiftly at a rate of 1000 devices per minute. Each job is `concurrent` and counts towards the job concurrency limit only for a short time. After the job executions have been rolled out or canceled, the job is no longer concurrent and does not count towards the job concurrency limit. You can use the job concurrency to create a large number of jobs while waiting for devices to complete the job execution.

> **Note**
>
> If a job with the optional scheduling configuration and job document rollout scheduled to take place during a maintenance window reaches the selected `startTime` and you're at your maximum job concurrency limit, then that scheduled job will move to a status state of `CANCELED`.

To determine whether a job is concurrent, you can use the `IsConcurrent` property of a job from the AWS IoT console, or by using the `DescribeJob` or `ListJob` API. This limit applies to both continuous and snapshot jobs.

To view the active jobs and job concurrency limits and other AWS IoT Jobs quotas for your AWS account and to request a limit increase, see AWS IoT Device Management endpoints and quotas in the AWS General Reference.

The following diagram shows how the job concurrency applies to in-progress jobs and jobs that are being canceled.



> **Note**
>
> New jobs with the optional `SchedulingConfig` will maintain an initial status state of `SCHEDULED` and update to `IN_PROGRESS` upon reaching the selected `startTime`. After the new job with the optional `SchedulingConfig` reaches the selected `startTime` and updates to `IN_PROGRESS`, it will count towards the active jobs limit and job concurrency limit. Jobs with a status state of `SCHEDULED` will count towards the active jobs limit, but will not count towards the job concurrency limit.

The following table shows the limits that apply to active and concurrent jobs and the concurrent and non-concurrent phases of the job states.

Active and concurrent job limits

Job status	Phase	Active jobs limit	Job concurrency limit
`SCHEDULED`	Non-concurrent phase: AWS IoT Jobs waits for the scheduled `startTime` of the job to begin job execution notifications to your devices. Jobs in this phase only count towards the active jobs limit and will have the `IsConcurrent` property set to false.	Applies	Does not apply
`IN_PROGRESS`	Concurrent phase: AWS IoT Jobs accepts the request for creating the job and starts rolling out job execution notifications to your devices. Jobs in this phase	Applies	Applies

Job status	Phase	Active jobs limit	Job concurrency limit
	are concurrent, as denoted by the `IsConcurrent` property set to true, and count towards both the active jobs and the job concurrency limits.		
	Non-concurrent phase: AWS IoT Jobs waits for devices to report the results of their job executions. Jobs in this phase only count towards the active jobs limit and will have the `IsConcurrent` property set to false.	Applies	Does not apply
`Canceled`	Concurrent phase: AWS IoT Jobs accepts the request for canceling the job and starts canceling job executions previousl y created for your devices. Jobs in this phase are concurrent and will have the `IsConcurrent` property set to true. Once the job and job executions have been canceled, the job is no longer concurrent and does not count towards the job concurrency limit.	Does not apply	Applies

Note

The max duration of a recurring maintenance window is 23 hours, 50 minutes.

AWS IoT Device Management commands

> **Important**
>
> This documentation describes how you can use the commands feature in AWS IoT Device Management. For information about using this feature for AWS IoT FleetWise, see Remote commands.
> You are solely responsible for deploying commands in a manner that is safe and compliant with applicable laws. For more information on your responsibilities, please see the AWS Service Terms for AWS IoT Services.

Use AWS IoT Device Management Commands to send an instruction from the cloud to a device that's connected to AWS IoT. Commands target one device at a time and can be used for low-latency, high-throughput applications, such as retrieving device-side logs or initiating a device state change.

The *command* is a reusable resource that's managed by AWS IoT Device Management. It contains configurations that are applied before they are published to the device. You can predefine a set of commands for specific use cases, such as turning on a light bulb or unlocking a vehicle door.

The AWS IoT Commands feature enables you to:

- Create reusable command templates with static or dynamic payloads, then execute them on specific devices.
- Target devices registered as AWS IoT things or unregistered MQTT clients.
- Run multiple commands concurrently on the same device.
- Enable event notifications and track execution status as devices process commands.

The following topics show you how to create commands, send them to your device, and retrieve the status reported by the device.

Topics

- Create and manage commands
- Start and monitor command executions
- Deprecate a command resource

Quick start

Complete these steps to send your first command:

1. **Prerequisites** - Ensure you are onboarded to IoT Core, a device connected to IoT Core, IAM permissions for Commands API operations, and an MQTT client library installed on your device.
2. **Create a command** - Define a command with a static payload, or a payload template specifying device actions. See Create a command resource.
3. **Identify your device** - Register your device as an IoT thing or note its MQTT client ID. See Target device considerations.
4. **Configure permissions** - Set up IAM policies allowing your device to receive commands and publish responses.
5. **Subscribe to topics** - Configure your device to subscribe to commands request and response topics. See Choose target device for your commands and subscribe to MQTT topics.
6. **Execute the command** - Start the command execution on your target device. See Start a command execution.

Common use cases:

- *Remote diagnostics* - Retrieve logs, run diagnostics, check device health
- *Configuration updates* - Change settings, update parameters, toggle features
- *Device control* - Lock/unlock, power on/off, mode changes

Commands concepts and status

Use AWS IoT Commands to send instructions from the cloud to connected devices. To use this feature:

1. Create a command with a payload containing the configurations required to run on the device.

2. Specify the target device that will receive the payload and perform the actions.
3. Execute the command on the target device and retrieve status information. To troubleshoot issues, see CloudWatch logs.

For more information about this workflow, see High-level commands workflow.

Topics

Commands key concepts

The following key concepts help you understand the Commands feature. Terms are used consistently throughout this documentation:

- *Command* - A reusable template defining device instructions
- *Execution* - An instance of a command running on a device
- *Thing name* - Identifier for devices registered in IoT registry
- *Client ID* - MQTT identifier for unregistered devices
- *Payload* - The instruction data sent to devices
- *Topic* - MQTT channel for command communication

Commands

Commands are instructions sent from the cloud to your IoT devices as MQTT messages. After receiving the payload, devices process the instructions and take corresponding actions, such as modifying configuration settings, transmitting sensor readings, or uploading logs. Devices then return results to the cloud, enabling remote monitoring and control.

Namespace

When creating a command, specify its namespace. For AWS IoT Device Management commands, use the default `AWS-IoT` namespace and provide either a payload or payloadTemplate. For AWS IoT FleetWise commands, use the `AWS-IoT-FleetWise` namespace. For more information, see Remote commands in the *AWS IoT FleetWise Developer Guide*.

Payload

When creating a command, provide a static payload that defines the actions the device must perform. The payload can use any supported format. To ensure devices correctly interpret the payload, we recommend specifying the payload format type. Devices using the MQTT5 protocol can follow the MQTT standard to identify the format. Format indicators for JSON or CBOR are available in the commands request topic.

Payload template

A payload template defines a command payload with placeholders that generate different payloads at runtime based on parameter values you provide. For example, instead of creating separate payloads for different temperature values, create one template with a temperature placeholder and specify the value during execution. This eliminates maintaining multiple similar payloads.

Target device

To execute a command, specify a target device using either its thing name (for devices registered with AWS IoT) or MQTT client ID (for unregistered devices). The client ID is a unique identifier defined in the MQTT protocol used to connect devices to AWS IoT. For details, see Target device considerations.

Commands topics

Before executing a command, devices must subscribe to the commands request topic. When you execute a command, the payload is sent to the device on this topic. After execution, devices publish results and status to the commands response topic. For more information, see Commands topics.

Command execution

An execution is an instance of a command running on a target device. When you start an execution, the payload is delivered to the device and a unique execution ID is generated. The device executes the command and reports progress to AWS IoT. Device-side logic determines execution behavior and status reporting to reserved topics.

Parameter value conditions

When creating commands with payload templates, define value conditions to validate parameter values before execution. Value conditions ensure parameters meet requirements, preventing invalid executions.

Supported operators by CommandParameterValue type

Numeric types (INTEGER, LONG, DOUBLE, UNSIGNEDLONG)

- `EQUALS` - Value must equal the specified number
- `NOT_EQUALS` - Value must not equal the specified number
- `GREATER_THAN` - Value must be greater than the specified number
- `GREATER_THAN_EQUALS` - Value must be greater than or equal to the specified number
- `LESS_THAN` - Value must be less than the specified number
- `LESS_THAN_EQUALS` - Value must be less than or equal to the specified number
- `IN_RANGE` - Value must be within the specified range (inclusive)
- `NOT_IN_RANGE` - Value must be outside the specified range (inclusive)
- `IN_SET` - Value must match one of the specified numbers
- `NOT_IN_SET` - Value must not match any of the specified numbers

String type (STRING)

- `EQUALS` - Value must equal the specified string
- `NOT_EQUALS` - Value must not equal the specified string
- `IN_SET` - Value must match one of the specified strings
- `NOT_IN_SET` - Value must not match any of the specified strings

Boolean type

- Value conditions are not supported

Binary type

- Value conditions are not supported

Example: Temperature control command

```
{
  "commandId": "SetTemperature",
  "namespace": "AWS-IoT",
  "payloadTemplate": "{\"temperature\":
 \"${aws:iot:commandexecution::parameter:temperature}\"}",
  "parameters": [
    {
      "name": "temperature",
      "type": "INTEGER",
```

```
      "valueConditions": [
        {
          "comparisonOperator": "IN_RANGE",
          "operand": {
            "numberRange": {
              "min": "60",
              "max": "80"
            }
          }
        }
      ]
    }
  ]
}
```

In this example, the `temperature` parameter must be between 60 and 80 (inclusive). Execution requests with values outside this range fail validation.

Note

Value conditions are evaluated at invocation of StartCommandExecution API. Failed validations return an error and prevent execution creation.

Parameter value priority and evaluation

When starting command executions with payload templates, parameter values are resolved using the following priority:

1. **Execution request parameters** - Values provided in the `StartCommandExecution` request take highest priority
2. **Command default values** - If a parameter is not provided in the execution request, the parameter's `defaultValue` is used
3. **No value** - If neither is provided, execution fails as the parameter required to generate the execution request

Value conditions are evaluated on the final parameter value derived above on the priority and before execution creation. If validation fails, the execution request returns an error.

Example: SetTemperature command with `defaultValue`

```
{
  "parameters": [
    {
      "name": "temperature",
      "type": "INTEGER",
      "defaultValue": {"I": 72},
      "valueConditions": [
        {
          "comparisonOperator": "IN_RANGE",
          "operand": {"numberRange": {"min": "60", "max": "80"}}
        }
      ]
    }
  ]
}
```

When starting execution:

- If you provide `"temperature": {"I": 75}` in the request, 75 is used
- If you omit the temperature parameter, the default value 72 is used
- Both values are validated against the range condition [60,80]

Command states

Commands in your AWS account can be in one of three states: *Available*, *Deprecated*, or *Pending deletion*.

Available

After successful creation, a command is in the Available state and can be executed on devices.

Deprecated

Mark commands for deprecation when no longer needed. Deprecated commands cannot start new executions, but pending executions continue to completion. To enable new executions, restore the command to Available state.

Pending deletion

When you mark a command for deletion, it is deleted automatically if deprecated longer than the maximum timeout (default: 12 hours). This action is permanent. If not deprecated or deprecated for less than the timeout, the command enters Pending deletion state and is removed after the timeout expires.

Command execution status

When you start an execution on a target device, it enters CREATED status and can transition to other statuses based on device reports. You can retrieve status information and track executions.

> **Note**
>
> You can run multiple commands concurrently on a device. Use concurrency control to limit executions per device and prevent overload. For maximum concurrent executions per device, see [AWS IoT Device Management commands quotas].

The following table shows execution statuses and their transitions based on execution progress.

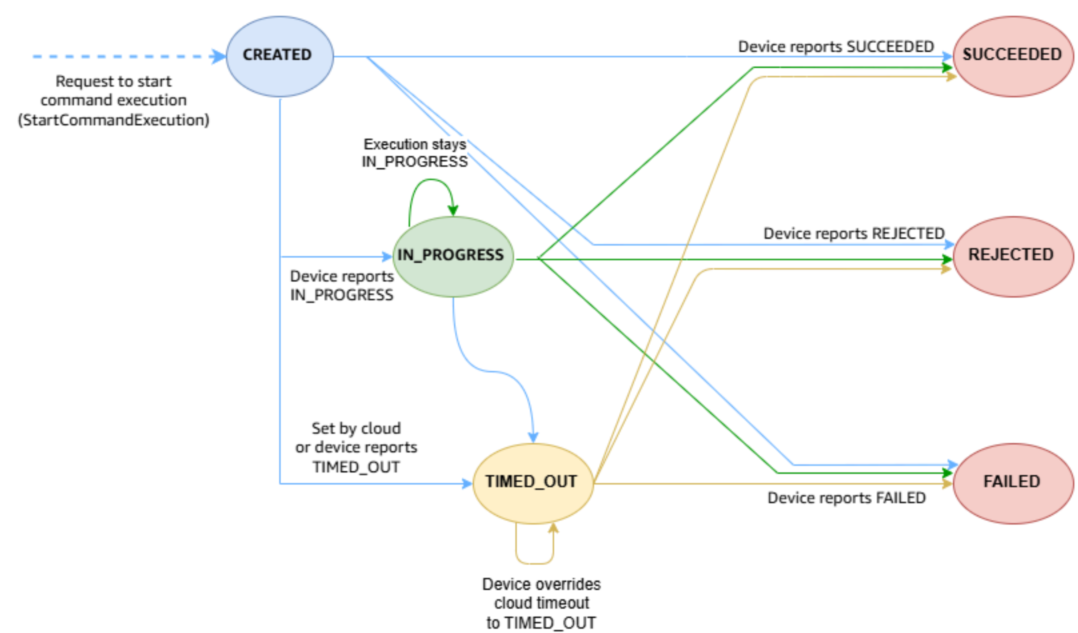
Command execution status and source

Command execution status	Initiated by device/cloud?	Terminal execution?	Allowed status transitions
CREATED	Cloud	No	• IN_PROGRESS • SUCCEEDED • FAILED • REJECTED • TIMED_OUT
IN_PROGRESS	Device	No	• IN_PROGRESS • SUCCEEDED • FAILED • REJECTED • TIMED_OUT

Command execution status	Initiated by device/cloud?	Terminal execution?	Allowed status transitions
`TIMED_OUT`	Device and cloud	No	• SUCCEEDED • FAILED • REJECTED • TIMED_OUT
`SUCCEEDED`	Device	Yes	Not applicable
`FAILED`	Device	Yes	Not applicable
`REJECTED`	Device	Yes	Not applicable

Devices can publish status and result updates anytime using commands reserved MQTT topics. To provide additional context, devices can use `reasonCode` and `reasonDescription` fields in the `statusReason` object.

The following diagram shows execution status transitions.



> **Note**
>
> When AWS IoT detects no device response within the timeout period, it sets `TIMED_OUT` as a temporary status allowing retries and state changes. If your device explicitly reports `TIMED_OUT`, this becomes a terminal status with no further transitions. For more information, see Non-terminal command executions.

The following sections describe terminal and non-terminal executions and their statuses.

Topics

Non-terminal command executions

An execution is non-terminal if it can accept updates from devices. Non-terminal executions are considered *Active*. The following statuses are non-terminal:

- **CREATED**

  When you start an execution from the AWS IoT console or use the `StartCommandExecution` API, successful requests change the status to `CREATED`. From this status, executions can transition to any other non-terminal or terminal status.

- **IN_PROGRESS**

  After receiving the payload, devices can start executing instructions and performing specified actions. While executing, devices can publish responses to the commands response topic and update status to `IN_PROGRESS`. From `IN_PROGRESS`, executions can transition to any terminal or non-terminal status except `CREATED`.

  > **Note**
  >
  > The `UpdateCommandExecution` API can be invoked multiple times with `IN_PROGRESS` status. Specify additional execution details using the `statusReason` object.

- **TIMED_OUT**

Both cloud and device can trigger this status. Executions in `CREATED` or `IN_PROGRESS` status can change to `TIMED_OUT` for the following reasons:

- After sending the command, a timer starts. If the device doesn't respond within the specified duration, the cloud changes status to `TIMED_OUT`. In this case, the execution is non-terminal.
- The device can override status to any terminal status or report a timeout and set status to `TIMED_OUT`. In this case, status remains `TIMED_OUT`, but `StatusReason` object fields change based on device information. The execution becomes terminal.

For more information, see Time out value and `TIMED_OUT` execution status.

Terminal command executions

An execution becomes terminal when it no longer accepts updates from devices. The following statuses are terminal. Executions can transition to terminal statuses from any non-terminal status: `CREATED`, `IN_PROGRESS`, or `TIMED_OUT`.

- **SUCCEEDED**

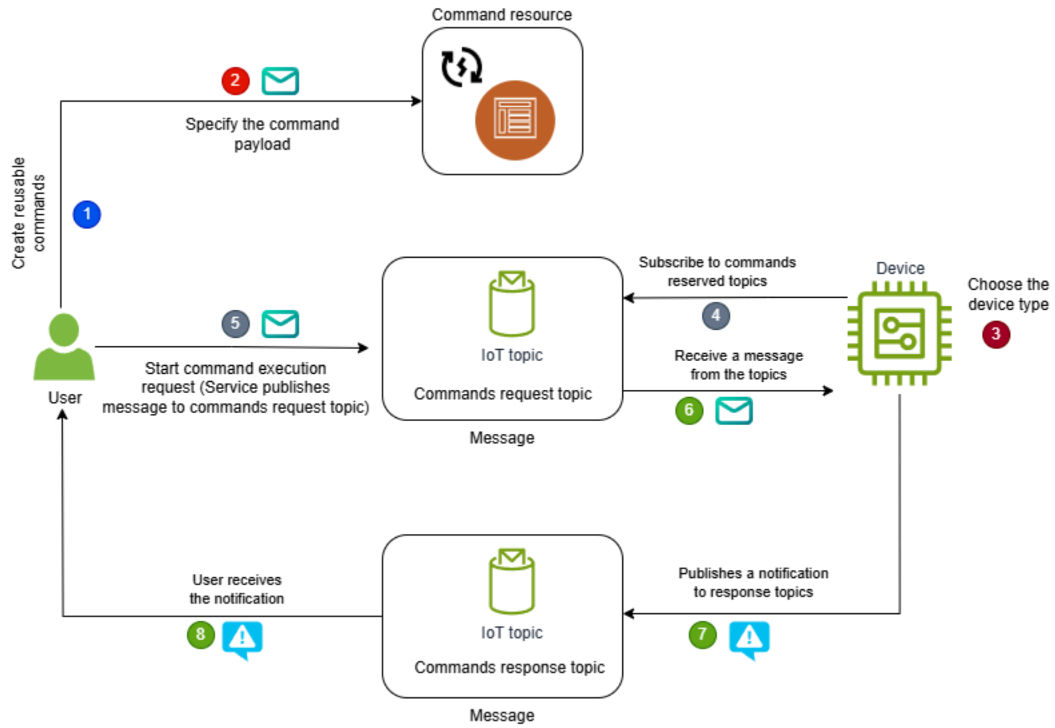
  If the device successfully completes execution, it can publish a response to the commands response topic and update status to `SUCCEEDED`.
- **FAILED**

  When a device fails to complete execution, it can publish a response to the commands response topic and update status to `FAILED`. Use the `reasonCode` and `reasonDescription` fields in the `statusReason` object, or CloudWatch logs, to troubleshoot failures.
- **REJECTED**

  When a device receives an invalid or incompatible request, it can invoke the `UpdateCommandExecution` API with status `REJECTED`. Use the `reasonCode` and `reasonDescription` fields in the `statusReason` object, or CloudWatch logs, to troubleshoot issues.

High-level commands workflow

This workflow shows how devices interact with AWS IoT Device Management Commands. All HTTP API requests use Sigv4 credentials for signing.



Workflow overview

- [Create and manage commands]
- [Choose target device for your commands and subscribe to MQTT topics]
- [Start and monitor command executions for your target device]
- [(Optional) Enable notifications for commands events]

Create and manage commands

To create and manage commands for your devices, perform the following steps.

1. **Create a command resource**

   Create a Command from the Command Hub or using the `CreateCommand` API.

2. **Specify the payload**

   Provide a Payload in any format. Specify the content type to ensure correct device interpretation.

For dynamic Commands with payload templates, the final payload is generated at runtime using your supplied parameters. Templates support JSON format only, but the generated Payload can be sent as JSON or CBOR.

3. **(Optional) Manage the created commands**

   Update the display name and description after creation. Mark Commands as deprecated when no longer needed, or delete them permanently. To modify Payload information, create a new Command.

Choose target device for your commands and subscribe to MQTT topics

Choose your target device and configure MQTT topics for receiving Commands and publishing responses.

1. **Choose the target device for your command**

   Choose a target device to receive and execute the Command. Use a Thing name for registered devices or a Client ID for unregistered devices. For more information, see Target device considerations.

2. **Configure the AWS IoT device policy**

   Configure an IAM policy granting permissions to receive Executions and publish updates. See Sample IAM policy for policy examples.

3. **Establish an MQTT connection**

   Connect devices to the message broker and subscribe to request and response Topics. Devices need `iot:Connect` permission. Find your data plane endpoint using the `DescribeEndpoint` API or `describe-endpoint` CLI command:

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   Running this command returns the account-specific data plane endpoint as shown below.

   ```
   account-specific-prefix.iot.region.amazonaws.com
   ```

4. **Subscribe to commands topics**

   Subscribe to the Commands request Topic. When you start an Execution, the message broker publishes the Payload to this Topic. Your device receives and processes the Command.

   (Optional) Subscribe to response Topics (`accepted` or `rejected`) to receive confirmation whether the cloud service accepted or rejected the device response.

   In this example, replace:

   - *<device>* with `thing` or `client` depending on whether the device you're targeting has been registered as an IoT thing, or specified as an MQTT client.
   - *<DeviceID>* with the unique identifier of your target device. This ID can be the unique MQTT client ID or a thing name.

   > **Note**
   >
   > If the payload type is not JSON or CBOR, the *<PayloadFormat>* field might not be present in the commands request topic. To get the payload format, we recommend that you use MQTT5 to get the format information from the MQTT message headers. For more information, see Commands topics.

   ```
   $aws/commands/<devices>/<DeviceID>/executions/+/request/<PayloadFormat>
   $aws/commands/<devices>/<DeviceID>/executions/+/response/accepted/<PayloadFormat>
   $aws/commands/<devices>/<DeviceID>/executions/+/response/rejected/<PayloadFormat>
   ```

Start and monitor command executions for your target device

After you have created the commands and specified the targets for the command, you can start the execution on the target device by performing the following steps.

1. **Start command execution on the target device**

   Start the Execution from the Command Hub or using the `StartCommandExecution` API with your account-specific endpoint. Use `iot:Data-ATS` for dual-stack (IPv4/IPv6) or `iot:Jobs` for IPv4 only.

The API publishes the Payload to the Commands request Topic.

> **Note**
>
> If the device is offline and uses MQTT persistent sessions, the Command waits at the message broker. When the device reconnects before timeout, it can process the Command and publish results. If timeout expires, the execution times out and the payload will be discarded.

2. **Update the result of the command execution**

   The device receives the Payload, processes the Command, performs the specified actions, and publishes results to the Commands response Topic using `UpdateCommandExecution` MQTT based API. If subscribed to accepted and rejected Topics, the device receives confirmation whether the cloud service accepted or rejected the response.

   Depending on what you specified in the request topic, *<devices>* can be either things or clients, and *<DeviceID>* can be your AWS IoT thing name or the MQTT client ID.

   > **Note**
   >
   > The *<PayloadFormat>* can only be JSON or CBOR in the commands response topic.

   ```
   $aws/commands/<devices>/<DeviceID>/executions/<ExecutionId>/
   response/<PayloadFormat>
   ```

3. **(Optional) Retrieve command execution result**

   Retrieve Execution results from the AWS IoT console or using `GetCommandExecution`. The device must publish results to the Commands response Topic for latest information. View additional details including last update time, result, and completion time.

(Optional) Enable notifications for commands events

Subscribe to Commands events for notifications when Execution status changes.

1. **Create a topic rule**

   Subscribe to the Commands events Topic for status change notifications. Create a topic rule to route device data to other AWS IoT services like AWS Lambda, Amazon SQS, and AWS Step Functions using the AWS IoT console or [Creating an AWS IoT rule](#).

   In this example, replace *<CommandID>* with the identifier of the command for which you want to receive notifications and *<CommandExecutionStatus>* with the status of the command execution.

   ```
   $aws/events/commandExecution/<CommandID>/<CommandExecutionStatus>
   ```

   > **Note**
   >
   > To receive notifications for all commands and command execution statuses, you can use wildcard characters and subscribe to the following topic.

   ```
   $aws/events/commandExecution/+/#
   ```

2. **Receive and process commands events**

   Manage Commands push notifications and build applications using the subscribed events.

The following code shows a sample payload for the commands events notifications that you'll receive.

```
{
    "executionId": "2bd65c51-4cfd-49e4-9310-d5cbfdbc8554",
    "status":"FAILED",
    "statusReason": {
         "reasonCode": "DEVICE_TOO_BUSY",
         "reasonDescription": ""
    },
    "eventType": "COMMAND_EXECUTION",
    "commandArn":"arn:aws:iot:us-east-1:123456789012:command/0b9d9ddf-
e873-43a9-8e2c-9fe004a90086",
    "targetArn":"arn:aws:iot:us-east-1:123456789012:thing/5006c3fc-
de96-4def-8427-7eee36c6f2bd",
```

```
    "timestamp":1717708862107
}
```

Create and manage commands

Use AWS IoT Device Management Commands to configure reusable remote actions or send immediate instructions to devices. Create and manage Commands from the AWS IoT console or using the AWS CLI.

Create and manage commands operations

- Create a command resource
- Retrieve information about a command
- List commands in your AWS account
- Update a command resource
- Deprecate or restore a command resource
- Delete a command resource

Create a command resource

Provide the following information when creating a Command:

- **General information**

  Provide a unique Command ID to identify the Command when running it on target devices. Optionally specify a display name, description, and tags for management.
- **Payload**

  For static Commands, provide a Payload defining device actions. Specify the Payload format type for correct device interpretation.

  For dynamic Commands, see the Payload template attribute.
- **Payload template**

  For dynamic Commands, provide a payloadTemplate with placeholders and parameters. Optionally provide `defaultValue` and conditions. AWS IoT Device Management Commands replaces placeholders at runtime. Missing parameters use their defaultValue. All values must satisfy defined conditions.

The following case-sensitive placeholder types are supported:

- `${aws:iot:commandexecution::parameter:parameter1}` – A placeholder for the value of a parameter with the name `parameter1`.
- `${aws:iot:commandexecution::executionTimeoutSec}` – A placeholder for the command execution timeout parameter supplied at runtime.

Payload and commands topics

Commands reserved Topics use a format based on Payload format type.

- For `application/json` or `application/cbor` content types, use this request Topic:

```
$aws/commands/<devices>/<DeviceID>/executions/+/request/<PayloadFormat>
```

- For other content types or unspecified format, use this request Topic. The Payload format appears in the MQTT message header.

```
$aws/commands/<devices>/<DeviceID>/executions/+/request
```

The Commands response Topic uses `json` or `cbor` format regardless of Payload type. *`<PayloadFormat>`* must be `json` or `cbor`:

```
$aws/commands/<devices>/<DeviceID>/executions/<ExecutionId>/response/<PayloadFormat>
```

Create a command resource (console)

The following sections describe Payload format considerations and Command creation from the console.

Topics

Static command payload format

The Payload supports any format up to 32 KB. Specify the Payload format type for secure and correct device interpretation.

Specify Payload format type using `type/subtype` format (e.g., `application/json` or `application/cbor`). Default: `application/octet-stream`. See [Common MIME types](#) for supported formats.

Dynamic command payload format

The payloadTemplate must be valid JSON with at least one placeholder, up to 32 KB.

For `AwsJsonSubstitution` preprocessor, AWS IoT Device Management Commands sends notifications in JSON or CBOR format based on preprocessor configuration.

How to create a command (console)

To create a Command from the console, go to [Command Hub](#) and follow these steps:

1. Choose **Create command**.
2. Specify a unique Command ID.
3. (Optional) Specify display name, description, and tags.
4. Upload the Payload file containing device actions. Specify the Payload format type for correct device interpretation.
5. (Optional) For JSON payload templates with placeholders, parameters pre-populate in the inline table for editing.
6. (Optional) Configure parameter value type (required), default value, and conditions.
7. Choose **Create command**.

Create a command resource (CLI)

Use the [`CreateCommand`](#) API or [`create-command`](#) CLI command to create a Command.

Topics

- Command payload
- Sample IAM policy
- Static command creation example

- Dynamic command creation example

Command payload

Provide a static Payload or payload template. Static Payloads are base64 encoded. For payload templates, the final Payload generates at runtime using parameter values. Devices process the Payload and perform specified actions. Specify the Payload content type for correct device reception.

> **Note**
>
> Payloads cannot be modified after Command creation. Create a new Command to modify the Payload.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `CreateCommand` action.

In this example, replace:

- *`region`* with your AWS Region, such as *`us-east-1`*.
- *`account-id`* with your AWS account number, such as *`123456789012`*.
- *`command-id`* with a unique identifier for your AWS IoT command ID, such as *`LockDoor`*. If you want to send more than one command, you can specify these commands under the *Resource* section in the IAM policy.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": "iot:CreateCommand",
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:command/command-id"
    }
}
```

Static command creation example

The following example shows how you can create a static command. Depending on your application, replace:

- *<command-id>* with a unique identifier for the command. For example, to lock the door of your house, you can specify *LockDoor*. We recommend that you use UUID. You can also use alpha-numeric characters, "-", and "_".
- (Optional) *<display-name>* and *<description>* , which are optional fields that you can use to provide a friendly name and a meaningful description for the command, such as *Lock the doors of my home*.
- `namespace`, which you can use to specify the namespace of the command. It must be `AWS-IoT`. For information about using this feature for AWS IoT FleetWise, see [Remote commands](#)
- `payload` contains information about the payload that you want to use when running the command and it's content type.

```
aws iot create-command \
    --command-id <command-id> \
    --display-name <display-name> \
    --description <description> \
    --namespace AWS-IoT \
    --payload
 '{"content":"eyAibWVzc2FnZSI6ICJIZWxsbyBJb1QiIH0=","contentType":"application/json"}'
```

Running this command generates a response that contains the ID and ARN (Amazon resource name) of the command. For example, if you specified the *LockDoor* command during creation, the following shows a sample output of running the command.

```
{
    "commandId": "LockDoor",
    "commandArn": "arn:aws:iot:us-east-1:123456789012:command/LockDoor"
}
```

Dynamic command creation example

The following example shows how you can create a dynamic command. Depending on your application, replace:

- *<command-id>* with a unique identifier for the command. For example, to set the light power status, you can specify *Light_Power_Status*. We recommend that you use UUID. You can also use alpha-numeric characters, "-", and "_".
- (Optional) *<display-name>* and *<description>* , which are optional fields that you can use to provide a friendly name and a meaningful description for the command, such as *Turn a light ON or OFF*.
- `namespace`, which you can use to specify the namespace of the command. It must be `AWS-IoT`. For information about using this feature for AWS IoT FleetWise, see [Remote commands]
- `payloadTemplate` contains the JSON formatted playoad template with placehodlers.
- `preprocessor` contains the configuration for preprocessor that determines how the payloadTemplate must be processed.
- `mandatoryParameter` contains the parameters corresponding to the placeholders in the payloadTemplate, their type, default values, and conditions.

```
aws iot create-command \
    --command-id Light_Power_Status \
    --description "Turn a light ON or OFF" \
    --namespace AWS-IoT \
    --payload-
template '{"powerStatus":"${aws:iot:commandexecution::parameter:powerStatus}"}' \
    --preprocessor awsJsonSubstitution={outputFormat=JSON} \
    --mandatory-parameters "name=powerStatus, defaultValue={S=OFF},
 valueConditions=[{comparisonOperator=IN_SET, operand={strings=['ON','OFF']}}]"
```

Running this command generates a response that contains the ID and ARN (Amazon resource name) of the command. For example, if you specified the *Light_Power_Status* command during creation, the following shows a sample output of running the command.

```
{
    "commandId": "Light_Power_Status",
    "commandArn": "arn:aws:iot:us-east-1:123456789012:command/Light_Power_Status"
}
```

Retrieve information about a command

After you create a command, you can retrieve information about it from the AWS IoT console and using the AWS CLI. You can obtain the following information.

- The command ID, Amazon resource name (ARN), any display name and description that you specified for the command.
- The command state, which indicates whether a command is available to run on the target device, or whether it's being deprecated or deleted.
- The payload or payloadTemplate that you provided.
- The preprocessor that you provided.
- The mandatoryParameters that you provided.
- The time when the command was created and last updated.

Retrieve a command resource (console)

To retrieve a command from the console, go to the Command Hub of the AWS IoT console and then choose the command that you created to view it's details.

In addition to the command details, you can see the command history, which provides information about the executions of the command on the target device. After you run this command on the device, you can find information about the executions on this tab.

Retrieve a command resource (CLI)

Use the `GetCommand` HTTP control plane API operation or the `get-command` AWS CLI command to retrieve information about a command resource. You must've already created the command using the `CreateCommand` API request or the `create-command` CLI.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `GetCommand` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as *`123456789023`*.

- *command-id* with your AWS IoT unique command identifier, such as *LockDoor* or *Light_Power_Status*. If you want to retrieve more than one command, you can specify these commands under the *Resource* section in the IAM policy.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": "iot:GetCommand",
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:command/command-id"
    }
}
```

Retrieve a command example (AWS CLI)

The following example shows you how to retrieve information about a command using the `get-command` AWS CLI. Depending on your application, replace *<command-id>* with the identifier for the command for which you want to retrieve information. You can obtain this information from the response of the `create-command` CLI.

```
aws iot get-command --command-id <command-id>
```

Running this command generates a response that contains information about the command, the payload, and the time when it was created and last updated. It also provides information that indicates whether a command has been deprecated or is being deleted.

For example, the following code shows a sample response.

```
{
    "commandId": "LockDoor",
    "commandArn": "arn:aws:iot:<region>:<account>:command/LockDoor",
    "namespace": "AWS-IoT",
    "payload":{
        "content": "eyAibWVzc2FnZSI6ICJIZWxsbyBJb1QiIH0=",
        "contentType": "application/json"
    },
    "createdAt": "2024-03-23T00:50:10.095000-07:00",
    "lastUpdatedAt": "2024-03-23T00:50:10.095000-07:00",
    "deprecated": false,
```

```
    "pendingDeletion": false
}
```

List commands in your AWS account

After you have created commands, you can view the commands that you have created in your account. In the list, you can find information about:

- The command ID, and any display name that you specified for the commands.
- The Amazon resource name (ARN) of the commands.
- The command state which indicates whether the commands are available to run on the target device, or whether they are deprecated.

> **Note**
>
> The list does not display that are being deleted from your account. If the commands are pending deletion, you can still view the details for these commands using their command ID.

- The time when the commands were created and last updated.

List commands in your account (console)

In the AWS IoT console, you can find the list of commands that you created and their details by going to the Command Hub.

List commands in your account (CLI)

To list the commands that you created, use the `ListCommands` API operation or the `list-commands` CLI.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `ListCommands` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.

- *account-id* with your AWS account number, such as *123456789012*.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": "iot:ListCommands",
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:command/*"
    }
}
```

List commands in your account example

The following command shows how to list commands in your account.

```
aws iot list-commands --namespace "AWS-IoT"
```

Running this command generates a response that contains a list of commands that you created, the time when the commands were created and when it was last updated. It also provides the command state information, which indicates whether a command has been deprecated or is available to run on the target device. For more information about the different statuses and status reason, see Command execution status.

Update a command resource

After you have created a command, you can update the display name and description of the command.

> **Note**
>
> The payload for the command can't be updated. To update this information or to use a modified payload, you'll need to create a new command.

Update a command resource (console)

To update a command from the console, go to the Command Hub of the AWS IoT console and perform the following steps.

1. To update an existing command resource, choose the command that you want to update, and then under **Actions**, choose **Edit**.
2. Specify the display name and description that you want to use, and any name-value pairs as tags for your command.
3. Choose **Edit** to save the command with the new settings.

Update a command resource (CLI)

Use the `UpdateCommand` control plane API operation or the `update-command` AWS CLI to update a command resource. Using this API, you can:

- Edit the display name and description of a command that you created.
- Deprecate a command resource, or restore a command that has already been deprecated.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `UpdateCommand` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as *`123456789012`*.
- *`command-id`* with your AWS IoT unique command identifier, such as *`LockDoor`*. If you want to retrieve more than one command, you can specify these commands under the *Resource* section in the IAM policy.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": "iot:UpdateCommand",
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:command/command-id"
    }
}
```

Update information about a command examples (AWS CLI)

The following example shows you how to update information about a command using the `update-command` AWS CLI command. For information about how you can use this API to deprecate or restore a command resource, see Update a command resource (CLI).

The example shows how you can update the display name and description of a command. Depending on your application, replace *<command-id>* with the identifier for the command for which you want to retrieve information.

```
aws iot update-command \
    --command-id <command-id>
    --displayname <display-name> \
    --description <description>
```

Running this command generates a response that contains the updated information about the command and the time when it was last updated. The following code shows a sample request and response for updating the display name and description of a command that turns off the AC.

```
aws iot update-command \
    --command-id <LockDoor> \
    --displayname <Secondary lock door> \
    --description <Locks doors to my home>
```

Running this command generates the following response.

```
{
    "commandId": "LockDoor",
    "commandArn": "arn:aws:iot:us-east-1:123456789012:command/LockDoor",
    "displayName": "Secondary lock door",
    "description": "Locks doors to my home",
    "lastUpdatedAt": "2024-05-09T23:15:53.899000-07:00"
}
```

Deprecate or restore a command resource

After you have created a command, if no longer want to continue using the command, you can mark it as deprecated. When you deprecate a command, all pending command executions will continue running on the target device until they reach a terminal status. Once a command has been deprecated, if you want to use it such as for sending a new command execution to the target device, you must restore it.

> **Note**
>
> You can't edit a deprecated command, or run any new executions for it. To run new commands on the device, you must restore it so that the command state changes to *Available*.

For additional information about deprecating and restoring a command, and considerations for it, see Deprecate a command resource.

Delete a command resource

If you no longer want to use a command, you can remove it permanently from your account. If the deletion action is successful:

- If the command has been deprecated for a duration that's longer than the maximum timeout of 12 hours, the command will be deleted immediately.
- If the command isn't deprecated, or has been deprecated for a duration shorter than the maximum timeout, the command will be in a `pending deletion` state. It will be removed automatically from your account after the maximum timeout of 12 hours.

> **Note**
>
> The command might be deleted even if there are any pending command executions. The command will be in a pending deletion state and will be removed from your account automatically.

Delete a command resource (console)

To delete a command from the console, go to the Command Hub of the AWS IoT console and perform the following steps.

1. Choose the command that you want to delete, and and then under **Actions**, choose **Delete**.
2. Confirm that you want to delete the command and then choose **Delete**.

The command will be marked for deletion and will be permanently removed from your account after 12 hours.

Delete a command resource (CLI)

Use the `DeleteCommand` HTTP control plane API operation or the `delete-command` AWS CLI command to delete a command resource. If the deletion action is successful, you'll see a HTTP `statusCode` of 204 or 202, and the command will be deleted from your account automatically after the maximum timeout duration of 12 hours. In the case of the 204 status, it indicates that the command has been deleted.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `DeleteCommand` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as *`123456789012`*.
- *`command-id`* with your AWS IoT unique command identifier, such as *`LockDoor`*. If you want to retrieve more than one command, you can specify these commands under the *Resource* section in the IAM policy.

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": "iot:DeleteCommand",
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:command/command-id"
    }
}
```

Delete a command example (AWS CLI)

The following examples show you how to delete a command using the `delete-command` AWS CLI command. Depending on your application, replace *<command-id>* with the identifier for the command that you're deleting.

```
aws iot delete-command --command-id <command-id>
```

If the API request is successful, then the command generates a status code of 202 or 204. You can use the `GetCommand` API to verify that the command no longer exists in your account.

Start and monitor command executions

After creating a Command, start an Execution on the target device. The device updates results and publishes status to MQTT reserved Topics. Retrieve and monitor Execution status from your account.

Start and monitor Commands using the AWS IoT console or AWS CLI.

Start and monitor commands operations

- Start a command execution
- Update the result of a command execution
- Retrieve a command execution
- Viewing commands updates using the MQTT test client
- List command executions in your AWS account
- Delete a command execution

Start a command execution

> **Important**
>
> You are solely responsible for deploying commands in a manner that is safe and compliant with applicable laws.

Before starting an Execution, ensure:

- You created a Command in the AWS IoT namespace with Payload information. When starting Execution, the device processes Payload instructions and performs specified actions. See Create a command resource for Command creation.
- Your device subscribed to MQTT reserved Topics for Commands. When starting Execution, Payload information publishes to this reserved MQTT request Topic:

  <devices> can be Things or MQTT clients. *<DeviceID>* is the Thing name or Client ID. Supported *<PayloadFormat>* values: JSON and CBOR. For more information, see Commands topics.

  ```
  $aws/commands/<devices>/<DeviceID>/executions/+/request/<PayloadFormat>
  ```

  For non-JSON/CBOR *<PayloadFormat>*, use this Commands Topic format:

  ```
  $aws/commands/<devices>/<DeviceID>/executions/+/request
  ```

Target device considerations

Specify the target device to receive and execute the Command. Use a Thing name for registered devices or Client ID for unregistered devices. After receiving the Payload, the device executes the Command and performs specified actions.

AWS IoT thing

Target devices can be Things registered in the AWS IoT registry. Things simplify device search and management.

Register devices as Things from the Connect device page or using `CreateThing`. Find existing Things from Thing Hub or using `DescribeThing`. See Managing things with the registry for registration details.

Client ID

For unregistered devices, use the Client ID.

The Client ID is a unique identifier you assign to devices. Defined in the MQTT protocol, it contains alphanumeric characters, underscores, or dashes. Each device connecting to AWS IoT needs a unique Client ID.

> **Note**
>
> - For registered Things, the Client ID can match the Thing name.
> - When targeting a specific Client ID, devices must connect to AWS IoT using that Client ID to receive the Payload.

The Client ID is the MQTT client ID devices use when connecting to AWS IoT Core. AWS IoT uses this ID to identify devices and manage connections and subscriptions.

Command execution timeout considerations

Timeout specifies the duration (in seconds) for devices to provide Execution results.

After creating an Execution, a timer starts. If the device goes offline or fails to report results within timeout, the Execution times out with status `TIMED_OUT`.

Default: 10 seconds. Maximum: 12 hours.

Time out value and `TIMED_OUT` execution status

Both cloud and device can report timeout.

After sending the Command, a timer starts. If no device response arrives within timeout, the cloud sets Execution status to `TIMED_OUT` with reason code `$NO_RESPONSE_FROM_DEVICE`.

This occurs when:

- Device went offline during Execution.
- Device failed to complete Execution within timeout.
- Device failed to report status within timeout.

In this instance, when the execution status of `TIMED_OUT` is reported from the cloud, the command execution is non-terminal. Your device can publish a response that overrides the status to any of the terminal statuses: `SUCCEEDED`, `FAILED`, or `REJECTED`. The command execution then becomes terminal and doesn't accept any further updates.

Your device can also update a `TIMED_OUT` status initiated by the cloud by reporting that a timeout occurred when it was executing the command. In this case, the command execution status remains at `TIMED_OUT`, but the `statusReason` object is updated based on the information reported by the device. The command execution then becomes terminal, and no further updates are accepted.

Using MQTT persistent sessions

You can configure MQTT persistent sessions to use with the AWS IoT Device Management commands feature. This feature is especially useful in cases such as when your device goes offline and you want to make sure that the device still receives the command when it comes back online before the timeout duration, and performs the instructions specified.

By default, the MQTT persistent session expiry is set to 60 minutes. If your command execution timeout is configured to a value that exceeds this duration, command executions that run longer than 60 minutes can be rejected by the message broker and fail. To run commands that are longer than 60 minutes in duration, you can request an increase to the persistent session expiry time.

> **Note**
>
> To ensure that you're using the MQTT persistent sessions feature correctly, set the Clean Start flag to zero. For more information, see MQTT persistent sessions.

Start a command execution (console)

To start running the command from the console, go to the Command Hub page of the AWS IoT console and perform the following steps.

1. To run the command that you've created, choose **Run command**.
2. Review information about the command that you created, including the MQTT reserved topics, and parameters, if applicable.

   For dynamic commands, enter the parameter values or leave them with defaults. For parameters that do not have a default value, you must provide a value to be sent as part of this execution.
3. Specify the target device to receive and execute the Command. The device can be specified as an AWS IoT thing if it has been registered with AWS IoT, or using the client ID if your device has not been registered yet. For more information, see Target device considerations
4. (Optional) Configure a timeout value for the command that determines the duration for which you want the command to run before it times out. If your command needs to run longer than

60 minutes, you may have to increase the MQTT persistent sessions expiry time. For more information, see Command execution timeout considerations.

5. Choose **Run command**.

Start a command execution (AWS CLI)

Use the `StartCommandExecution` HTTP data plane API operation to start a command execution. The API request and response are correlated by the command execution ID. After the device completes executing the command, it can report the status and execution result to the cloud by publishing a message to the commands response topic. For a custom response code, application codes that you own can process the response message and post the result to AWS IoT.

If your devices have subscribed to the commands request topic, the `StartCommandExecution` API will publish the payload message to the topic. The payload can use any format of your choice. For more information, see Command payload.

```
$aws/commands/<devices>/<DeviceID>/executions/+/request/<PayloadFormat>
```

If the payload format is not JSON or CBOR, the following shows the format of the commands request topic.

```
$aws/commands/<devices>/<DeviceID>/executions/+/request
```

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `StartCommandExecution` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as *`123456789012`*.
- *`command-id`* with a unique identifier for your AWS IoT command, such as *`LockDoor`*. If you want to send more than one command, you can specify these commands in the IAM policy.
- *`devices`* with either `thing` or `client` depending on whether your devices have been registered as AWS IoT things, or are specified as MQTT clients.
- *`device-id`* with your AWS IoT `thing-name` or `client-id`.

```
{
  "Effect": "Allow",
  "Action": [
      "iot:StartCommandExecution"
  ],
  "Resource": [
      "arn:aws:iot:region:account-id:command/command-id",
      "arn:aws:iot:region:account-id:devices/device-id"
  ]
}
```

To see a list of condition keys supported for `StartCommandExecution`, see Condition Keys for AWS IoT in the *IAM User Guide*.

Obtain account-specific data plane endpoint

Before you run the API command, you must obtain the account-specific endpoint URL for the endpoint. If you're using dual-stack endpoints (IPv4 and IPv6), use the `iot:Data-ATS`. The `iot:Jobs` endpoint is for IPv4 only. For example, if you run this command:

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

It returns the account-specific endpoint URL as shown in the sample response below.

```
{
    "endpointAddress":
    "<account-specific-prefix>-ats.iot.<region>.api.com"
}
```

Start a command execution example (AWS CLI)

The following example displays how to start executing a command using the `start-command-execution` AWS CLI command.

In this example, replace:

- *`<command-arn>`* with the ARN for the command that you want to execute. You can obtain this information from the response of the `create-command` CLI command. For example, if you're executing the command for changing the steering wheel mode, use `arn:aws:iot:`*`region`*`:`*`account-id`*`:command/`*`SetComfortSteeringMode`*.

- *<target-arn>* with the Thing ARN for the target device, which can be an IoT thing or MQTT client, for which you want to execute the command. For example, if you're executing the command for the target device `myRegisteredThing`, use `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`myRegisteredThing`*.
- *<endpoint-url>* with the account-specific endpoint that you obtained in Obtain account-specific data plane endpoint, prefixed by `https://`. For example, `https://`*`123456789012abcd`*`.jobs.iot.`*`us-east-1`*`.amazonaws.com`.
- (Optional) You can also specify an additional parameter, `executionTimeoutSeconds`, when performing the `StartCommandExecution` API operation. This optional field specifies the time in seconds within which the device must complete executing the command. By default, the value is 10 seconds. When the command execution status is `CREATED`, a timer starts. If the command execution result is not received before the timer expires, then the status automatically changes to `TIMED_OUT`.
-
  ```
  aws iot-jobs-data start-command-execution \
      --command-arn <command-arn>  \
      --target-arn <target-arn> \
      --endpoint <endpoint-url> \
      --execution-timeout-seconds 900
  ```
- (Optional) For dynamic commands, specify the parameters and their values to be used for substitution. You must provide a value for parameters that do not have a defaultValue set at command creation. If a parameter has a defaultValue, the parameter value provided here takes precedence. For parameters that have valueConditions set, the parameter value provided here must satisfy the condition.

  Based on `Light_Power_Status` dynamic command example:
-
  ```
  aws iot-jobs-data start-command-execution \
      --command-arn arn:aws:iot:us-east-1:123456789012:command/Light_Power_Status  \
      --target-arn arn:aws:iot:us-east-1:123456789012:thing/exampleThing \
      --endpoint <endpoint-url> \
      --execution-timeout-seconds 900 \
      --parameters "powerStatus={S=ON}"
  ```

Running this command returns a command execution ID. You can use this ID to query the command execution status, details, and command execution history.

> **Note**
>
> If the command has been deprecated, then the `StartCommandExecution` API request will fail with a validation exception. To fix this error, first restore the command using the `UpdateCommand` API, and then perform the `StartCommandExecution` request.

```
{
    "executionId": "07e4b780-7eca-4ffd-b772-b76358da5542"
}
```

Update the result of a command execution

Use the `UpdateCommandExecution` MQTT data plane API operation to update the status or result of a command execution.

> **Note**
>
> Before you use this API:
>
> - Your device must have established an MQTT connection and subscribed to the commands request and response topics. For more information, see High-level commands workflow.
> - You must have already executed this command using the `StartCommandExecution` API operation.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes your device to perform these actions. Following shows an example policy that authorizes your device to perform the action. For additional sample IAM policies that allow the user permission to perform the `UpdateCommandExecution` MQTT action, see Connect and publish policy examples.

In this example, replace:

- *`Region`* with your AWS Region, such as `us-east-1`.
- *`AccountID`* with your AWS account number, such as *`123456789012`*.

- *ThingName* with the name of your AWS IoT thing for which you are targeting the command execution, such as *myRegisteredThing*.
- *commands-request-topic* and *commands-response-topic* with the names of your AWS IoT commands request and response topics. For more information, see High-level commands workflow.

Sample IAM policy for MQTT client ID

The following code shows a sample device policy when using MQTT client ID.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response/json"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/request",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response/accepted",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response/rejected",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/request/json",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response/accepted/json",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/clients/
${iot:ClientId}/executions/*/response/rejected/json"
      ]
    },
```

```
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/request",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/response/accepted",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/response/rejected",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/request/json",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/response/accepted/json",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/clients/
${iot:ClientId}/executions/+/response/rejected/json"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:us-east-1:123456789012:client/${iot:ClientId}"
    }
  ]
}
```

Sample IAM policy for IoT thing

The following code shows a sample device policy when using an AWS IoT thing.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/response"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
```

```
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/request",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/response/accepted",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/response/rejected",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/request/json",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/response/accepted/json",
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/*/response/rejected/json"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/request",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/response/accepted",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/response/rejected",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/request/json",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/response/accepted/json",
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/commands/things/
${iot:Connection.Thing.ThingName}/executions/+/response/rejected/json"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:us-east-1:123456789012:client/${iot:ClientId}"
    }
  ]
}
```

How to use the UpdateCommandExecution API

After the command execution is received at the request topic, the device processes the command. It then uses the UpdateCommandExecution API to update the status and result of the command execution to the following response topic.

```
$aws/commands/<devices>/<DeviceID>/executions/<ExecutionId>/response/<PayloadFormat>
```

In this example, *<DeviceID>* is the unique identifier of your target device, and *<execution-id>* is the identifier of the command execution on the target device. The *<PayloadFormat>* can be JSON or CBOR.

> **Note**
>
> If you haven't registered your device with AWS IoT, you can use the client ID as your identifier instead of a thing name.

```
$aws/commands/clients/<ClientID>/executions/<ExecutionId>/response/<PayloadFormat>
```

Device reported updates to execution status

Your devices can use the API to report any of the following status updates to the command execution. For more information about these statuses, see Command execution status.

- IN_PROGRESS: When the device starts executing the command, it can update the status to IN_PROGRESS.
- SUCCEEDED: When the device successfully processes the command and completes executing it, the device can publish a message to the response topic as SUCCEEDED.
- FAILED: If the device failed to execute the command, it can publish a message to the response topic as FAILED.
- REJECTED: If the device failed to accept the command, it can publish a message to the response topic as REJECTED.
- TIMED_OUT: The command execution status can change to TIMED_OUT due to any of the following reasons.

- The result of the command execution wasn't received. This can happen because the execution wasn't completed within the specified duration, or if the device failed to publish the status information to the response topic.
- The device reports that a time out occurred when attempting to execute the command.

For more information about the `TIMED_OUT` status, see [Time out value and `TIMED_OUT` execution status](#).

Considerations when using the `UpdateCommandExecution` API

The following are some important considerations when using the `UpdateCommandExecution` API.

- Your devices can use an optional `statusReason` object to provide additional information about the execution. If your devices provide this object, the `reasonCode` field of the object is required, but the `reasonDescription` field is optional.
- When your devices use the `statusReason` object, the `reasonCode` must use the pattern `[A-Z0-9_-]+` and not exceed 64 characters in length. If you provide the `reasonDescription`, ensure that it doesn't exceed 1,024 characters in length. It can use any characters except control characters such as newlines.
- Your devices can use an optional `result` object to provide information about the result of the command execution, such as the return value of a remote function call. If you provide the `result`, it must require at least one entry.
- In the `result` field, you specify the entries as key-value pairs. For each entry, you must specify the data type information as a string, boolean, or binary. A string data type must use the key `s`, a boolean data type uses the key `b`, and a binary data type must use the key `bin`. Ensure that these keys are lowercase.
- If you encounter an error when running the `UpdateCommandExecution` API, you can view the error in the `AWSIoTLogsV2` log group in Amazon CloudWatch. For information about enabling logging and viewing the logs, see [Configure AWS IoT logging](#).

`UpdateCommandExecution` API example

The following code shows an example of how your device can use the `UpdateCommandExecution` API to report the execution status, the `statusReason` field to provide additional information about the status, and the result field to provide information about the result of the execution, such as the car battery percentage in this case.

```
{
  "status": "IN_PROGRESS",
  "statusReason": {
    "reasonCode": "200",
    "reasonDescription": "Execution_in_progress"
  },
  "result": {
        "car_battery": {
            "s": "car battery at 50 percent"
        }
    }
}
```

Retrieve a command execution

After you run a command, you can retrieve information about the command execution from the AWS IoT console and using the AWS CLI. You can obtain the following information.

> **Note**
>
> To retrieve the latest command execution status, your device must publish the status information to the response topic using the `UpdateCommandExecution` MQTT API, as described below. Until the device publishes to this topic, the `GetCommandExecution` API will report the status as `CREATED` or `TIMED_OUT`.

Each command execution that you create will have:

- An **Execution ID**, which is a unique identifier of the command execution.
- The **Status** of the command execution. When you run the command on the target device, the command execution enters a `CREATED` state. It can then transition to other command execution statuses as described below.
- The **Result** of the command execution.
- The unique **Command ID** and the target device for which executions have been created.
- The **Start date**, which shows the time when the command execution was created.

Retrieve a command execution (console)

You can retrieve a command execution from the console using either of the following methods.

- **From the Command hub page**

  Go to the [Command Hub](#) page of the AWS IoT console and perform these steps.

  1. Choose the command for which you created an execution on the target device.
  2. In the command details page, on the **Command history** tab, you'll see the executions that you created. Choose the execution for which you want to retrive information.
  3. If your devices used the `UpdateCommandExecution` API to provide the result information, you can then find this information in the **Results** tab of this page.

- **From the Thing hub page**

  If you chose an AWS IoT thing as your target device when running the command, you can view the execution details from the Thing hub page.

  1. Go to the [Thing Hub](#) page in the AWS IoT console and choose the thing for which you created the command execution.
  2. In the thing details page, on the **Command history**, you'll see the executions that you created. Choose the execution for which you want to retrive information.
  3. If your devices used the `UpdateCommandExecution` API to provide the result information, you can then find this information in the **Results** tab of this page.

Retrieve a command execution (CLI)

Use the `GetCommandExecution` AWS IoT Core control plane HTTP API operation to retrieve information about a command execution. You must have already executed this command using the `StartCommandExecution` API operation.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `GetCommandExecution` action.

In this example, replace:

- *region* with your AWS Region, such as `us-east-1`.
- *account-id* with your AWS account number, such as *123456789012*.
- *command-id* with your unique AWS IoT command identifier, such as *LockDoor*.
- *devices* with either `thing` or `client` depending on whether your devices have been registered as AWS IoT things, or are specified as MQTT clients.
- *device-id* with your AWS IoT `thing-name` or `client-id`.

```
{
  "Effect": "Allow",
  "Action": [
      "iot:GetCommandExecution"
  ],
  "Resource": [
      "arn:aws:iot:region:account-id:command/command-id",
      "arn:aws:iot:region:account-id:devices/device-id"
  ]
}
```

Retrieve a command execution example

The following example shows you how to retrieve information about a command that was executed using the `start-command-execution` AWS CLI command. The following example shows how you can retrieve information about a command that was executed to turn off the steering wheel mode.

In this example, replace:

- *<execution-id>* with the identifier for the command execution for which you want to retrieve information.
- *<target-arn>* with the Amazon Resource Number (ARN) of the device for which you're targeting the execution. You can obtain this information from the response of the `start-command-execution` CLI command.
- Optionally, if your devices used the `UpdateCommandExecution` API to provide the execution result, you can specify whether to include the command execution result in the response of the `GetCommandExecution` API using the `GetCommandExecution` API.

```
aws iot get-command-execution
```

```
    --execution-id <execution-id> \
    --target-arn <target-arn> \
    --include-result
```

Running this command generates a response that contains information about the ARN of the command execution, the execution status, and the time when it started executing, and when it completed. It also provides a `statusReason` object that contains additional information about the status. For more information about the different statuses and status reason, see Command execution status.

The following code shows a sample response from the API request.

> **Note**
>
> The `completedAt` field in the execution response corresponds to the time when the device reports a terminal status to the cloud. In the case of `TIMED_OUT` status, this field will be set only when the device reports a time out. When the `TIMED_OUT` status is set by the cloud, the `TIMED_OUT` status is not updated. For more information about the time out behavior, see Command execution timeout considerations.


```
{
    "executionId": "07e4b780-7eca-4ffd-b772-b76358da5542",
    "commandArn": "arn:aws:iot:us-east-1:123456789012:command/LockDoor",
    "targetArn": "arn:aws:iot:us-east-1:123456789012:thing/myRegisteredThing",
    "status": "SUCCEEDED",
    "statusReason": {
        "reasonCode": "DEVICE_SUCCESSFULLY_EXECUTED",
        "reasonDescription": "SUCCESS"
    },
    "result": {
        "sn": { "s": "ABC-001" },
        "digital": { "b": true }
    },
    "createdAt": "2024-03-23T00:50:10.095000-07:00",
    "completedAt": "2024-03-23T00:50:10.095000-07:00"
}
```

Viewing commands updates using the MQTT test client

You can use the MQTT test client to view the message exchange over MQTT when using the commands feature. After your device establishes an MQTT connection with AWS IoT, you can create a command, specify the payload, and then run it on the device. When you run the command, if your device subscribed to the MQTT reserved request topic for commands, it sees the payload message published to this topic.

The device then receives the payload instructions and performs the specified operations on the AWS IoT device. It then uses the `UpdateCommandExecution` API to publish the command execution result and status information to the MQTT reserved response topics for commands. AWS IoT Device Management listens to updates on the response topics and stores the updated information and publishes logs to AWS CloudTrail and Amazon CloudWatch. You can then retrieve the latest command execution information from the console or by using the `GetCommandExecution` API.

The following steps show how to use the MQTT test client to observe messages.

1. Open the MQTT test client in the AWS IoT console.
2. On the **Subscribe** tab, enter the following topic and then choose **Subscribe**, where *<thingId>* is the thing name of the device that you have registered with AWS IoT.

   Note

   You can find the thing name for your device from the Thing Hub page of the AWS IoT console. If you haven't registered your device as a thing, you can register the device when connecting to AWS IoT from the Connect device page.

   ```
   $aws/commands/things/<thingId>/executions/+/request
   ```

3. (Optional) On the **Subscribe** tab, you can also enter the following topics and choose **Subscribe**.

   ```
   $aws/commands/things/+/executions/+/response/accepted/json
   $aws/commands/things/+/executions/+/response/rejected/json
   ```

4. When you start a command execution, the message payload will be sent to the device using the request topic that the device has subscribed to, `$aws/commands/things/<thingId>/`

executions/+/request. In the MQTT test client, you should see the command payload that contains the instructions for the device to process the command.

5. After the device starts executing the command, it can publish status updates to the following MQTT reserved response topic for commands.

```
$aws/commands/<devices>/<device-id>/executions/<executionId>/response/json
```

For example, consider a command that you executed to turn on the AC of your car to reduce the temperature to a desired value. The following JSON shows a sample message that the vehicle published to the response topic which shows that it failed to execute the command.

```
{
  "deviceId": "My_Car",
  "executionId": "07e4b780-7eca-4ffd-b772-b76358da5542",
  "status": "FAILED",
  "statusReason": {
    "reasonCode": "CAR_LOW_ON_BATTERY",
    "reasonDescription": "Car battery is lower than 5 percent"
  }
}
```

In this case, you can charge your car's battery and then run the command again.

List command executions in your AWS account

After you run a command, you can retrieve information about the command execution from the AWS IoT console and using the AWS CLI. You can obtain the following information.

- An **Execution ID**, which is a unique identifier of the command execution.
- The **Status** of the command execution. When you run the command on the target device, the command execution enters a CREATED state. It can then transition to other command execution statuses as described below.
- The unique **Command ID** and the target device for which executions have been created.
- The **Start date**, which shows the time when the command execution was created.

List command executions in your account (console)

You can see all the command executions from the console using either of the following methods.

- **From the Command hub page**

  Go to the [Command Hub](#) page of the AWS IoT console and perform these steps.

  1. Choose the command for which you created an execution on the target device.
  2. In the command details page, go to the **Command history** tab, and you'll see a list of executions that you created.

- **From the Thing hub page**

  If you chose an AWS IoT thing as your target device when running the command, and created multiple command executions for a single device, you can view the executions for the device from the Thing hub page.

  1. Go to the [Thing Hub](#) page in the AWS IoT console and choose the thing for which you created the executions.
  2. In the thing details page, on the **Command history**, you'll see a list of executions that you created for the device.

List command executions in your account (CLI)

Use the `ListCommandExecutions` AWS IoT Core control plane HTTP API operation to list all command executions in your account.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes you to perform this action on the device. The following example shows an IAM policy that allows the user permission to perform the `ListCommandExecutions` action.

In this example, replace:

- *`region`* with your AWS Region, such as `us-east-1`.
- *`account-id`* with your AWS account number, such as *`123456789012`*.
- *`command-id`* with your unique AWS IoT command identifier, such as *`LockDoor`*.

```
{
  "Effect": "Allow",
  "Action": "iot:ListCommandExecutions",
  "Resource": *
}
```

List command executions example

The following example shows you how to list command executions in your AWS account.

When running the command, you must specify whether to filter the list to display only command executions that were created for a particular device using the `targetArn`, or executions for a particular command specified using the `commandArn`.

In this example, replace:

- *<target-arn>* with the Amazon Resource Number (ARN) of the device for which you're targeting the execution, such as `arn:aws:iot:`*`us-east-1`*`:`*`123456789012`*`:thing/`*`b8e4157c98f332cffb37627f`*.
- *<target-arn>* with the Amazon Resource Number (ARN) of the device for which you're targeting the execution, such as `arn:aws:iot:`*`us-east-1`*`:`*`123456789012`*`:thing/`*`b8e4157c98f332cffb37627f`*.
- *<after>* with the time after which you want to list the executions that were created, for example, `2024-11-01T03:00`.

```
aws iot list-command-executions \
--target-arn <target-arn> \
--started-time-filter '{after=<after>}' \
--sort-order "ASCENDING"
```

Running this command generates a response that contains a list of command executions that you created, and the time when the executions started executing, and when it completed. It also provides status information, and the `statusReason` object that contains additional information about the status.

```
{
    "commandExecutions": [
        {
            "commandArn": "arn:aws:iot:us-east-1:123456789012:command/TestMe002",
```

```
            "executionId": "b2b654ca-1a71-427f-9669-e74ae9d92d24",
            "targetArn": "arn:aws:iot:us-east-1:123456789012:thing/
b8e4157c98f332cffb37627f",
            "status": "TIMED_OUT",
            "createdAt": "2024-11-24T14:39:25.791000-08:00",
            "startedAt": "2024-11-24T14:39:25.791000-08:00"
        },
        {
            "commandArn": "arn:aws:iot:us-east-1:123456789012:command/TestMe002",
            "executionId": "34bf015f-ef0f-4453-acd0-9cca2d42a48f",
            "targetArn": "arn:aws:iot:us-east-1:123456789012:thing/
b8e4157c98f332cffb37627f",
            "status": "IN_PROGRESS",
            "createdAt": "2024-11-24T14:05:36.021000-08:00",
            "startedAt": "2024-11-24T14:05:36.021000-08:00"
        }
    ]
}
```

For more information about the different statuses and status reason, see Command execution status.

Delete a command execution

If you no longer want to use a command execution, you can remove it permanently from your account.

Note

- A command execution can be deleted only if it has entered a terminal status, such as `SUCCEEDED`, `FAILED`, or `REJECTED`.
- This operation can be performed only using the AWS IoT Core API or the AWS CLI. It is not available from the console.

Sample IAM policy

Before you use this API operation, make sure that your IAM policy authorizes your device to perform these actions. Following shows an example policy that authorizes your device to perform the action.

In this example, replace:

- *Region* with your AWS Region, such as `us-east-1`.
- *AccountID* with your AWS account number, such as *123456789012*.
- *CommandID* with the identifier of the command for which you want to delete the execution.
- *devices* with either `thing` or `client` depending on whether your devices have been registered as AWS IoT things, or are specified as MQTT clients.
- *device-id* with your AWS IoT `thing-name` or `client-id`.

```
{
  "Effect": "Allow",
  "Action": [
      "iot:DeleteCommandExecution"
  ],
  "Resource": [
      "arn:aws:iot:region:account-id:command/command-id",
      "arn:aws:iot:region:account-id:devices/device-id"
  ]
}
```

Delete a command execution example

The following example shows you how to delete a command using the `delete-command` AWS CLI command. Depending on your application, replace *<execution-id>* with the identifier for the command execution that you're deleting, and the *<target-arn>* with the ARN of your target device.

```
aws iot delete-command-execution \
--execution-id <execution-id> \
--target-arn <target-arn>
```

If the API request is successful, then the command execution generates a status code of 200. You can use the `GetCommandExecution` API to verify that the command execution no longer exists in your account.

Deprecate a command resource

Deprecate commands to indicate they are outdated and should not be used. For example, deprecate commands no longer actively maintained or when creating newer commands with the same ID but different payloads.

Key considerations

Important considerations when deprecating commands:

- Deprecating a command does not delete it. You can retrieve the command using its ID and restore it for reuse.
- Attempting to start new executions on deprecated commands generates an error, preventing use of outdated commands.
- To execute a deprecated command, first restore it. After restoration, the command becomes available for regular use and execution on target devices.
- If you deprecate a command while executions are in progress, they continue running until completion. You can still retrieve execution status.

Deprecate a command resource (console)

To deprecate a command from the console, go to the Command Hub of the AWS IoT console and perform the following steps.

1. Choose the command that you want to deprecate, and then under **Actions**, choose **Deprecate**.
2. Confirm that you want to deprecate the command and then choose **Deprecate**.

Deprecate a command resource (CLI)

Mark commands as deprecated using the `update-command` CLI. You must deprecate a command before deletion. To use a deprecated command, restore it first.

```
aws iot update-command \
    --command-id <command-id> \
    --deprecated
```

For example, if you deprecated the *ACSwitch* command that you updated in the example above, the following code shows a sample output of running the command.

```
{
    "commandId": "turnOffAc",
    "deprecated": true,
    "lastUpdatedAt": "2024-05-09T23:16:51.370000-07:00"
}
```

Check deprecation time and status

Use the GetCommand API to determine if a command is deprecated and when it was last deprecated.

```
aws iot get-command --command-id <turnOffAC>
```

This command generates a response containing command information, including creation and deprecation timestamps from the last updated field. This helps determine command lifetime and whether to delete or reuse it. The following shows a sample response for the *turnOffAc* command:

```
{
    "commandId": "turnOffAC",
    "commandArn": "arn:aws:iot:us-east-1:123456789012:command/turnOffAC",
    "namespace": "AWS-IoT",
    "payload": {
        "content": "testPayload.json",
        "contentType": "application/json"
    },
    "createdAt": "2024-03-23T00:50:10.095000-07:00",
    "lastUpdatedAt": "2024-05-09T23:16:51.370000-07:00",
    "deprecated": false
}
```

Restore a command resource

To use or send the ACSwitch command to your device, restore it first.

To restore a command from the console, go to the Command Hub of the AWS IoT console, choose the command that you want to restore, and then under **Actions**, choose **Restore**.

To restore a command using the AWS IoT Core API or the AWS CLI, use the `UpdateCommand` API operation or the `update-command` CLI. The following code shows a sample request and response.

```
aws iot update-command \
    --command-id <command-id>
    --no-deprecated
```

The following code shows a sample output.

```
{
    "commandId": "ACSwitch",
    "deprecated": false,
    "lastUpdatedAt": "2024-05-09T23:17:21.954000-07:00"
}
```

AWS IoT secure tunneling

When devices are deployed behind restricted firewalls at remote sites, you need a way to gain access to those devices for troubleshooting, configuration updates, and other operational tasks. Use secure tunneling to establish bidirectional communication to remote devices over a secure connection that is managed by AWS IoT. Secure tunneling does not require updates to your existing inbound firewall rules, so you can keep the same security level provided by firewall rules at a remote site.

For example, a sensor device located at a factory that is a couple hundred miles away is having trouble measuring the factory temperature. You can use secure tunneling to open and quickly start a session to that sensor device. After you have identified the problem (for example, a bad configuration file), you can reset the file and restart the sensor device through the same session. Compared to a more traditional troubleshooting (for example, sending a technician to the factory to investigate the sensor device), secure tunneling decreases incident response and recovery time and operational costs.

What is secure tunneling?

Use secure tunneling to access devices that are deployed behind port-restricted firewalls at remote sites. You can connect to the destination device from your laptop or desktop computer as the source device by using the AWS Cloud. The source and destination communicate by using an open source local proxy that runs on each device. The local proxy communicates with the AWS Cloud by using an open port that is allowed by firewall, typically 443. Data that is transmitted through the tunnel is encrypted using Transported Layer Security (TLS).

Topics

Secure tunneling concepts

The following terms are used by secure tunneling when establishing communication with remote devices. For information about how secure tunneling works, see How secure tunneling works.

Client access token (CAT)

A pair of tokens generated by secure tunneling when a new tunnel is created. The CAT is used by the source and destination devices to connect to the secure tunneling service. The CAT can only be used once to connect to the tunnel. To reconnect to the tunnel, rotate the client access tokens using the RotateTunnelAccessToken API operation or the rotate-tunnel-access-token CLI command.

Client token

A unique value generated by the client that AWS IoT secure tunneling can use for all subsequent retry connections to the same tunnel. This field is optional. If the client token is not provided, then the client access token (CAT) can only be used once for the same tunnel. Subsequent connection attempts using the same CAT will be rejected. For more information about using client tokens, see the local proxy reference implementation in GitHub.

Destination application

The application that runs on the destination device. For example, the destination application can be an SSH daemon for establishing an SSH session using secure tunneling.

Destination device

The remote device you want to access.

Device agent

An IoT application that connects to the AWS IoT device gateway and listens for new tunnel notifications over MQTT. For more information, see IoT agent snippet.

Local proxy

A software proxy that runs on the source and destination devices and relays a data stream between secure tunneling and the device application. The local proxy can be run in source mode or destination mode. For more information, see Local proxy.

Source device

The device an operator uses to initiate a session to the destination device, usually a laptop or desktop computer.

Tunnel

A logical pathway through AWS IoT that enables bidirectional communication between a source device and destination device.

How secure tunneling works

The following shows how secure tunneling establishes a connection between your source and destination device. For information about the different terms such as client access token (CAT), see Secure tunneling concepts.

1. **Open a tunnel**

   To open a tunnel for initiating a session with your remote destination device, you can use the AWS Management Console, the AWS CLI open-tunnel command, or the OpenTunnel API.

2. **Download the client access token pair**

   After you've opened a tunnel, you can download the client access token (CAT) for your source and destination and save it on your source device. You must retrieve the CAT and save it now before starting the local proxy.

3. **Start local proxy in destination mode**

   The IoT agent that has been installed and is running on your destination device will be subscribed to the reserved MQTT topic `$aws/things/thing-name/tunnels/notify` and will receive the CAT. Here, *`thing-name`* is the name of the AWS IoT thing you create for your destination. For more information, see Secure tunneling topics.

   The IoT agent then uses the CAT to start the local proxy in destination mode and set up a connection on the destination side of the tunnel. For more information, see IoT agent snippet.

4. **Start local proxy in source mode**

   After the tunnel has been opened, AWS IoT Device Management provides the CAT for the source that you can download on the source device. You can use the CAT to start the local proxy in source mode, which then connects the source side of the tunnel. For more information about local proxy, see Local proxy.

5. **Open an SSH session**

   As both sides of the tunnel are connected, you can start an SSH session by using the local proxy on the source side.

For more information about how to use the AWS Management Console to open a tunnel and start an SSH session, see Open a tunnel and start SSH session to remote device.

The following video describes how secure tunneling works and walks you through the process of setting up an SSH session to a Raspberry Pi device.

Secure tunnel lifecycle

Tunnels can have the status `OPEN` or `CLOSED`. Connections to the tunnel can have the status `CONNECTED` or `DISCONNECTED`. The following shows how the different tunnel and connection statuses work.

1. When you open a tunnel, it has a status of `OPEN`. The tunnel's source and destination connection status is set to `DISCONNECTED`.
2. When a device (source or destination) connects to the tunnel, the corresponding connection status changes to `CONNECTED`.
3. When a device disconnects from the tunnel while the tunnel status remains `OPEN`, the corresponding connection status changes back to `DISCONNECTED`. A device can connect to and disconnect from a tunnel repeatedly as long as the tunnel remains `OPEN`.

> **Note**
>
> The client access tokens (CAT) can only be used once to connect to a tunnel. To reconnect to the tunnel, rotate the client access tokens using the RotateTunnelAccessToken API operation or the rotate-tunnel-access-token CLI command. For examples, see Resolving AWS IoT secure tunneling connectivity issues by rotating client access tokens.

4. When you call `CloseTunnel` or the tunnel remains `OPEN` for longer than the `MaxLifetimeTimeout` value, a tunnel's status becomes `CLOSED`. You can configure `MaxLifetimeTimeout` when calling `OpenTunnel`. `MaxLifetimeTimeout` defaults to 12 hours if you do not specify a value.

> **Note**
>
> A tunnel cannot be reopened when it is `CLOSED`.

5. You can call `DescribeTunnel` and `ListTunnels` to view tunnel metadata while the tunnel is visible. The tunnel can be visible in the AWS IoT console for at least three hours before it is deleted.

AWS IoT secure tunneling tutorials

AWS IoT secure tunneling helps customers establish bidirectional communication to remote devices that are behind a firewall over a secure connection managed by AWS IoT.

To demo AWS IoT secure tunneling, use our AWS IoT secure tunneling demo on GitHub.

The following tutorials will help you learn how to get started and use secure tunneling. You'll learn how to:

1. Create a secure tunnel using the quick setup and manual setup methods for accessing the remote device.
2. Configure the local proxy when using the manual setup method and connect to the tunnel to access the destination device.
3. SSH into the remote device from a browser without having to configure the local proxy.
4. Convert a tunnel created using the AWS CLI or using the manual setup method to use the quick setup method.

Tutorials in this section

The tutorials in this section focus on creating a tunnel using the AWS Management Console and the AWS IoT API Reference. In the AWS IoT console, you can create a tunnel from the Tunnels hub page or from the details page of a thing that you created. For more information, see Tunnel creation methods in AWS IoT console.

Following shows the tutorials in this section:

- **Open a tunnel and use browser-based SSH to access remote device**

  This tutorial shows how to open a tunnel from the Tunnels hub page using the quick setup method. You'll also learn how to use browser-based SSH to access the remote device using an in-context command line interface within the AWS IoT console.

- **Open a tunnel using manual setup and connect to remote device**

  This tutorial shows how to open a tunnel from the Tunnels hub page using the manual setup method. You'll also learn how to configure and start the local proxy from a terminal in your source device and connect to the tunnel.

- **Open a tunnel for remote device and use browser-based SSH**

  This tutorial shows how to open a tunnel from the details page of a thing that you created. You'll learn how to create a new tunnel and use an existing tunnel. The existing tunnel corresponds to the most recent, open tunnel that was created for the device. You can also use the browser-based SSH to access the remote device.

AWS IoT secure tunneling tutorials

- Open a tunnel and start SSH session to remote device
- Open a tunnel for remote device and use browser-based SSH

Open a tunnel and start SSH session to remote device

In these tutorials, you'll learn how to remotely access a device that's behind a firewall. You can't start a direct SSH session into the device because the firewall blocks all inbound traffic. The tutorials show you how you can open a tunnel and then use that tunnel to start an SSH session to a remote device.

Prerequisites for the tutorials

The prerequisites for running the tutorial can vary depending on whether you use the manual or quick setup methods for opening a tunnel and accessing the remote device.

> **Note**
>
> For both setup methods, you must allow outbound traffic on port 443.

- For information about prerequisites for the quick setup method tutorial, see Prerequisites for quick setup method.
- For information about prerequisites for the manual setup method tutorial, see Prerequisites for manual setup method. If you use this setup method, you must configure the local proxy on your source device. To download the local proxy source code, see Local proxy reference implementation on GitHub.

Tunnel setup methods

In these tutorials, you'll learn about the manual and quick setup methods for opening a tunnel and connecting to the remote device. The following table shows the difference between the setup methods. After you create the tunnel, you can use an in-browser command line interface to SSH into the remote device. If you misplace the tokens or the tunnel gets disconnected, you can send new access tokens to reconnect to the tunnel.

Quick and manual setup methods

Criteria	Quick setup	Manual setup
Tunnel creation	Create a new tunnel with default, editable configurations. To access your remote device, you can only use SSH as the destination service.	Create a tunnel by manually specifying the tunnel configurations. You can use this method to connect to the remote device using services other than SSH.
Access tokens	The destination access token will be automatically delivered to your device on the reserved MQTT topic, if a thing name is specified when creating the tunnel. You don't have to download or manage the token on your source device.	You'll have to manually download and manage the token on your source device. The destination access token is automatic ally delivered to the remote device on the reserved MQTT topic, if a thing name is specified when creating the tunnel.
Local proxy	A web-based local proxy is automatic ally configured for you for interacti ng with the device. You don't have to manually configure the local proxy.	You'll have to manually configure and launch the local proxy. To configure the local proxy, you can either use the AWS IoT Device Client or download the Local proxy reference implementation on GitHub.

Tunnel creation methods in AWS IoT console

The tutorials in this section show you how to create a tunnel using the AWS Management Console and the OpenTunnel API. If you configure the destination when creating a tunnel, AWS IoT secure tunneling delivers the destination client access token to the remote device over MQTT and the reserved MQTT topic, `$aws/things/RemoteDeviceA/tunnels/notify`). On receiving the

MQTT message, the IoT agent on the remote device starts the local proxy in destination mode. For more information, see Reserved topics.

> **Note**
>
> You can omit the destination configuration if you want to deliver the destination client access token to the remote device through another method. For more information, see Configuring a remote device and using IoT agent.

In the AWS IoT console, you can create a tunnel using either of the following methods. For information about tutorials that will help you learn to create a tunnel using these methods, see Tutorials in this section.

- **Tunnels hub**

  When you create the tunnel, you'll be able to specify whether to use the quick setup or the manual setup methods for creating the tunnel and provide the optional tunnel configuration details. The configuration details also include the name of the destination device and the service that you want to use for connecting to the device. After you create a tunnel, you can either SSH within the browser or open a terminal outside the AWS IoT console to access your remote device.
- **Thing details page**

  When you create the tunnel, you'll also be able to specify whether to use the most recent, open tunnel or create a new tunnel for the device, in addition to choosing the setup methods and providing any optional tunnel configuration details. You can't edit the configuration details of an existing tunnel. You can use the quick setup method to rotate the access tokens and SSH into the remote device within the browser. To open a tunnel using this method, you must have created an IoT thing (for example, `RemoteDeviceA`) in the AWS IoT registry. For more information, see Register a device in the AWS IoT registry.

Tutorials in this section

- Open a tunnel and use browser-based SSH to access remote device
- Open a tunnel using manual setup and connect to remote device

Open a tunnel and use browser-based SSH to access remote device

You can use the quick setup or the manual setup method for creating a tunnel. This tutorial shows how to open a tunnel using the quick setup method and use the browser-based SSH to connect to the remote device. For an example that shows how to open a tunnel using the manual setup method, see Open a tunnel using manual setup and connect to remote device.

Using the quick setup method, you can create a new tunnel with default configurations that can be edited. A web-based local proxy is configured for you and the access token is automatically delivered to your remote destination device using MQTT. After creating a tunnel, you can start interacting with your remote device using a command line interface within the console.

With the quick setup method, you must use SSH as the destination service to access the remote device. For more information about the different setup methods, see Tunnel setup methods.

Prerequisites for quick setup method

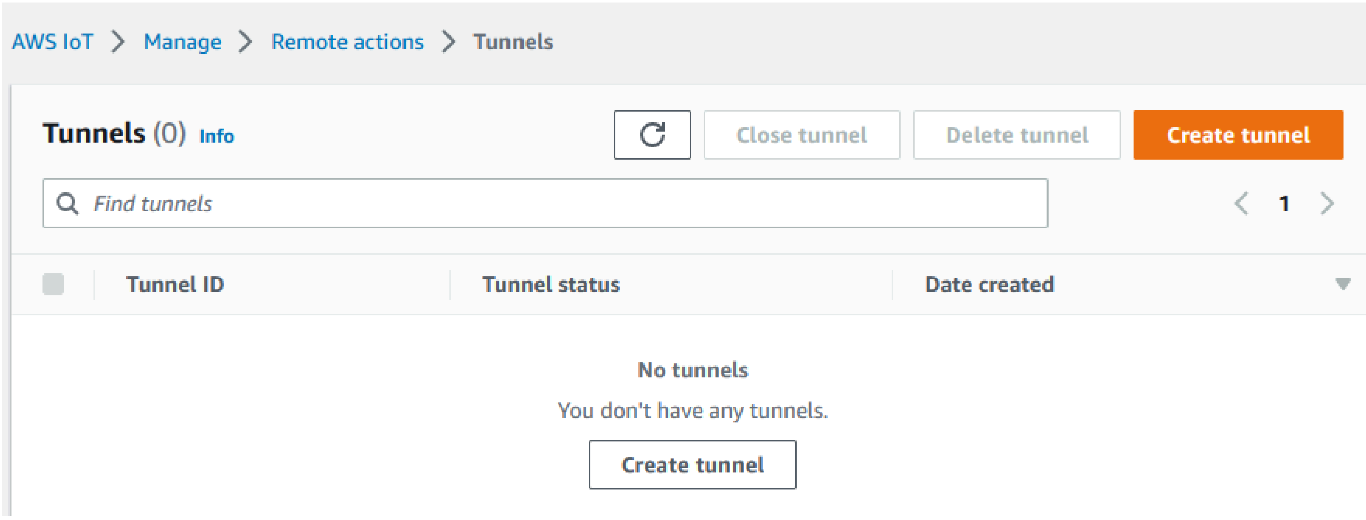
- The firewalls that the remote device is behind must allow outbound traffic on port 443. The tunnel that you create will use this port to connect to the remote device.
- You have an IoT device agent (see IoT agent snippet) running on the remote device that connects to the AWS IoT device gateway and is configured with an MQTT topic subscription. For more information, see connect a device to the AWS IoT device gateway.
- You must have an SSH daemon running on the remote device.

Open a tunnel

You can open a secure tunnel using the AWS Management Console, the AWS IoT API Reference, or the AWS CLI. You can optionally configure a destination name but it's not required for this tutorial. If you configure the destination, secure tunneling will automatically deliver the access token to the remote device using MQTT. For more information, see Tunnel creation methods in AWS IoT console.

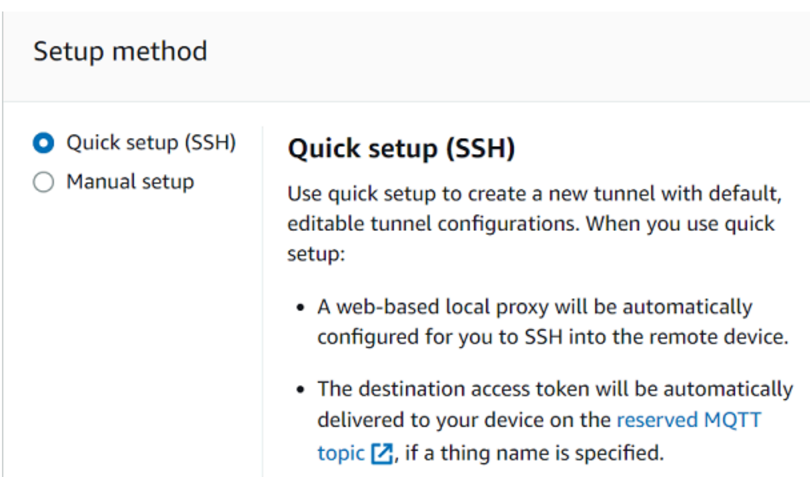
To open a tunnel using the console

1. Go to the Tunnels hub of the AWS IoT console and choose **Create tunnel**.



2. For this tutorial, choose **Quick setup** as the tunnel creation method and then choose **Next**.

> **Note**
>
> If you create a secure tunnel from the details page of a thing you created, you can choose whether to create a new tunnel or use an existing one. For more information, see Open a tunnel for remote device and use browser-based SSH.

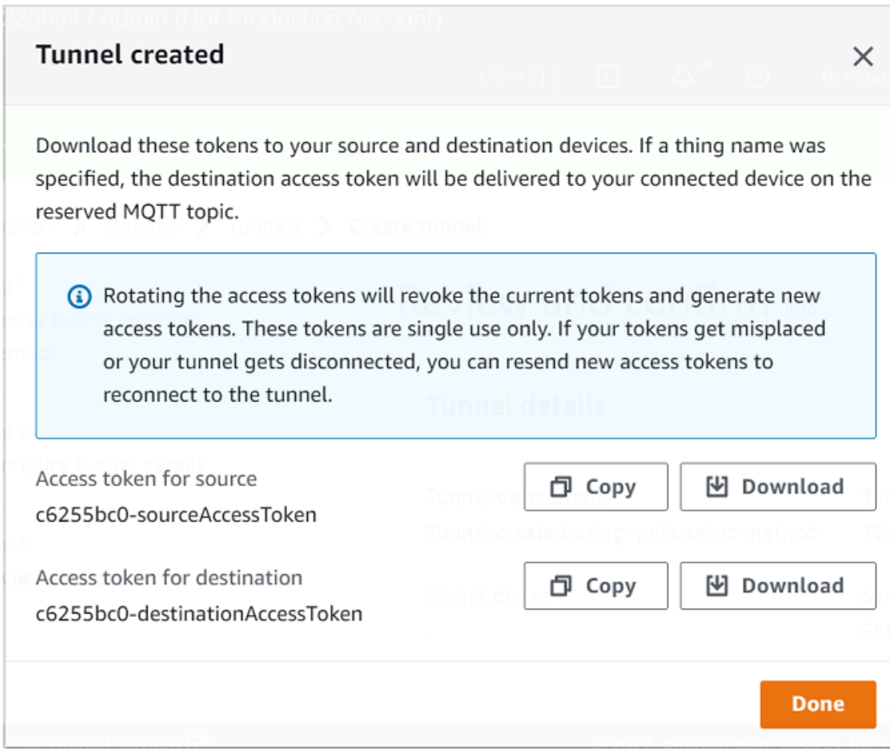


3. Review and confirm the tunnel configuration details. To create a tunnel, choose **Confirm and create**. If you want to edit these details, choose **Previous** to go back to the previous page and then confirm and create the tunnel.

> **Note**
>
> When using quick setup, the service name can't be edited. You must use **SSH** as the **Service**.

4. To create the tunnel, choose **Done**.

   For this tutorial, you don't have to download the source or destination access tokens. These tokens can only be used once to connect to the tunnel. If your tunnel gets disconnected, you can generate and send new tokens to your remote device for reconnecting to the tunnel. For more information, see Resend tunnel access tokens.



To open a tunnel using the API

To open a new tunnel, you can use the OpenTunnel API operation.

> **Note**
>
> You can create a tunnel using the quick setup method only from the AWS IoT console. When you use the AWS IoT API Reference API or the AWS CLI, it will use the manual setup method. You can open the existing tunnel that you created and then change the setup

method of the tunnel to use the quick setup. For more information, see Open an existing tunnel and use browser-based SSH.

The following shows an example of how to run this API operation. Optionally, if you want to specify the thing name and the destination service, use the `DestinationConfig` parameter. For an example that shows how to use this parameter, see Open a new tunnel for the remote device.

```
aws iotsecuretunneling open-tunnel
```

Running this command creates a new tunnel and provides you the source and destination access tokens.

```
{
    "tunnelId": "01234567-89ab-0123-4c56-789a01234bcd",
    "tunnelArn": "arn:aws:iot:us-
east-1:123456789012:tunnel/01234567-89ab-0123-4c56-789a01234bcd",
    "sourceAccessToken": "<SOURCE_ACCESS_TOKEN>",
    "destinationAccessToken": "<DESTINATION_ACCESS_TOKEN>"
}
```

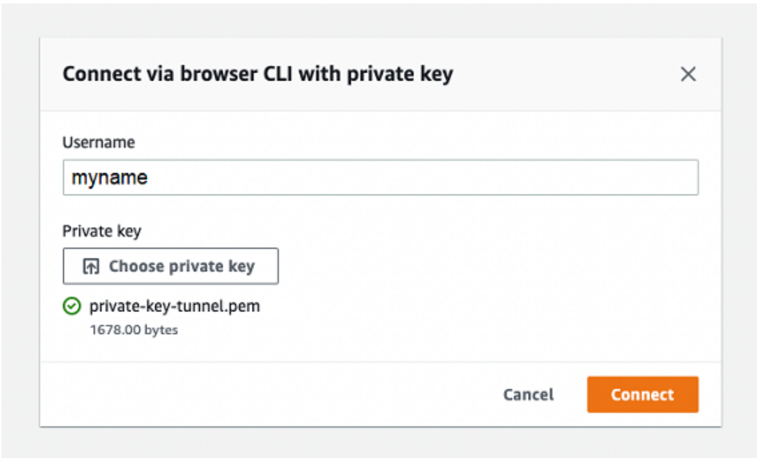
Using the browser-based SSH

After you create a tunnel using the quick setup method, and your destination device has connected to the tunnel, you can access the remote device using a browser-based SSH. Using the browser-based SSH, you can directly communicate with the remote device by entering commands into an in-context command line interface within the console. This feature makes it easier for you to interact with the remote device because you don't have to open a terminal outside the console or configure the local proxy.

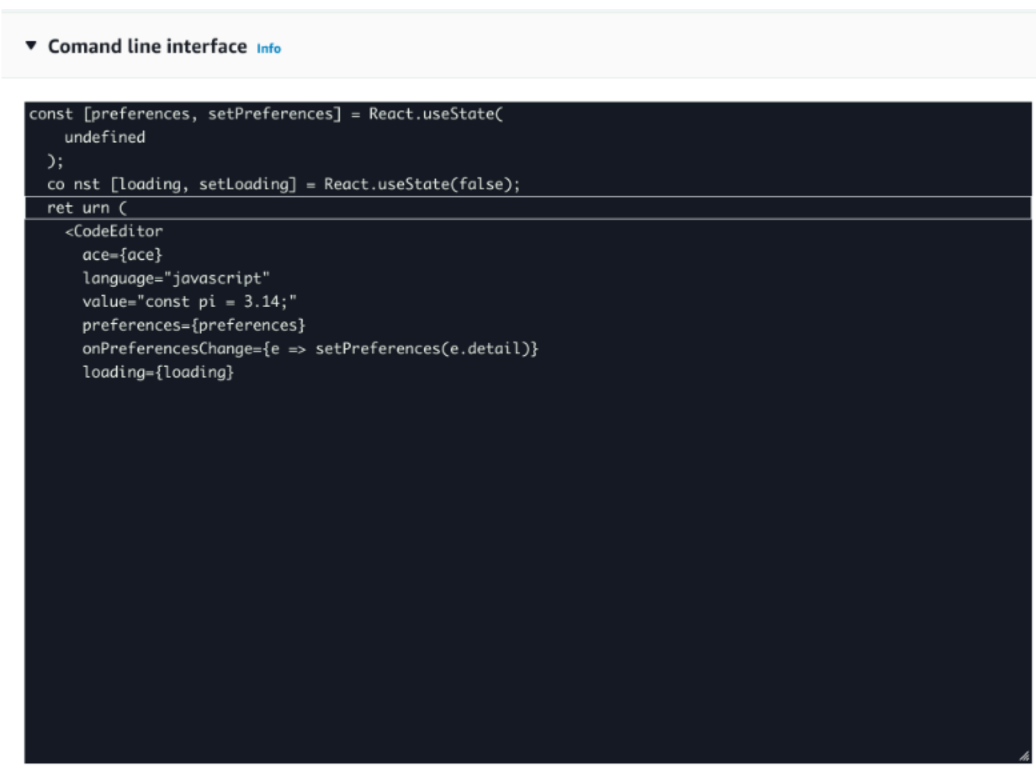
To use the browser-based SSH

1. Go to the Tunnels hub of the AWS IoT console and choose the tunnel that you created to view its details.
2. Expand the **Secure Shell (SSH)** section and then choose **Connect**.
3. Choose whether you want to authenticate into the SSH connection by providing your username and password, or, for more secure authentication, you can use your device's private key. If you're authenticating using the private key, note that only PEM formatted (256 and 512) RSA keys work with AWS IoT Secure Tunneling SSH console.

- To connect using your username and password, choose **Use password**. You can then enter your username and password and start using the in-browser CLI.
- To connect using your destination device's private key, choose **Use private key**. Specify your username and upload the device's private key file, and then choose **Connect** to start using the in-browser CLI.



After you've authenticated into the SSH connection, you can quickly get started with entering commands and interact with the device using the browser CLI, as the local proxy has already been configured for you.



If the browser CLI stays open after the tunnel duration, it might time out, causing the command line interface to get disconnected. You can duplicate the tunnel and start another session to interact with the remote device within the console itself.

Troubleshooting issues when using the browser-based SSH

The following shows how to troubleshoot some issues that you might run into when using the browser-based SSH.

- **You see an error instead of the command line interface**

  You might be seeing the error because your destination device got disconnected. You can choose **Generate new access tokens** to generate new access tokens and send the tokens to your remote device using MQTT. The new tokens can be used to reconnect to the tunnel. Reconnecting to the tunnel clears the history and refreshes the command line session.
- **You see a tunnel disconnected error when authenticating using private key**

  You might be seeing the error because your private key might not have been accepted by the destination device. To troubleshoot this error, check the private key file that you uploaded for authentication. If you still see an error, check your device logs. You can also try reconnecting to the tunnel by sending new access tokens to your remote device.
- **Your tunnel was closed when using the session**

  If your tunnel was closed because it stayed open for more than the specified duration, your command line session might get disconnected. A tunnel cannot be reopened once closed. To reconnect, you must open another tunnel to the device.

  You can duplicate a tunnel to create a new tunnel with the same configurations as the closed tunnel. You can duplicate a closed tunnel from the AWS IoT console. To duplicate the tunnel, choose the tunnel that was closed to view its details, and then choose **Duplicate tunnel**. Specify the tunnel duration that you want to use and then create the new tunnel.

Cleaning up

- **Close tunnel**

  We recommend that you close the tunnel after you've finished using it. A tunnel can also become closed if it stayed open for longer than the specified tunnel duration. A tunnel cannot be reopened once closed. You can still duplicate a tunnel by choosing the closed tunnel and then choosing **Duplicate tunnel**. Specify the tunnel duration that you want to use and then create the new tunnel.

- To close an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to close, and then choose **Close tunnel**.
- To close an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API.

```
aws iotsecuretunneling close-tunnel \
    --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
```

- **Delete tunnel**

  You can delete a tunnel permanently from your AWS account.

  > **Warning**
  >
  > Deletion actions are permanent and can't be undone.

- To delete an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to delete, and then choose **Delete tunnel**.
- To delete an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API. When using the API, set the `delete` flag to `true`.

```
aws iotsecuretunneling close-tunnel \
    --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
    --delete true
```

Open a tunnel using manual setup and connect to remote device

When you open a tunnel, you can choose the quick setup or the manual setup method for opening a tunnel into the remote device. This tutorial shows how to open a tunnel using the manual setup method and configure and start the local proxy to connect to the remote device.

When you use the manual setup method, you must manually specify the tunnel configurations when creating the tunnel. After creating the tunnel, you can SSH within the browser or open a terminal outside the AWS IoT console. This tutorial shows how to use the terminal outside the console to access the remote device. You'll also learn how to configure the local proxy and then connect to the local proxy to interact with the remote device. To connect to the local proxy, you must download the source access token when creating the tunnel.

With this setup method, you can use services other than SSH, such as FTP to connect to the remote device. For more information about the different setup methods, see Tunnel setup methods.

Prerequisites for manual setup method

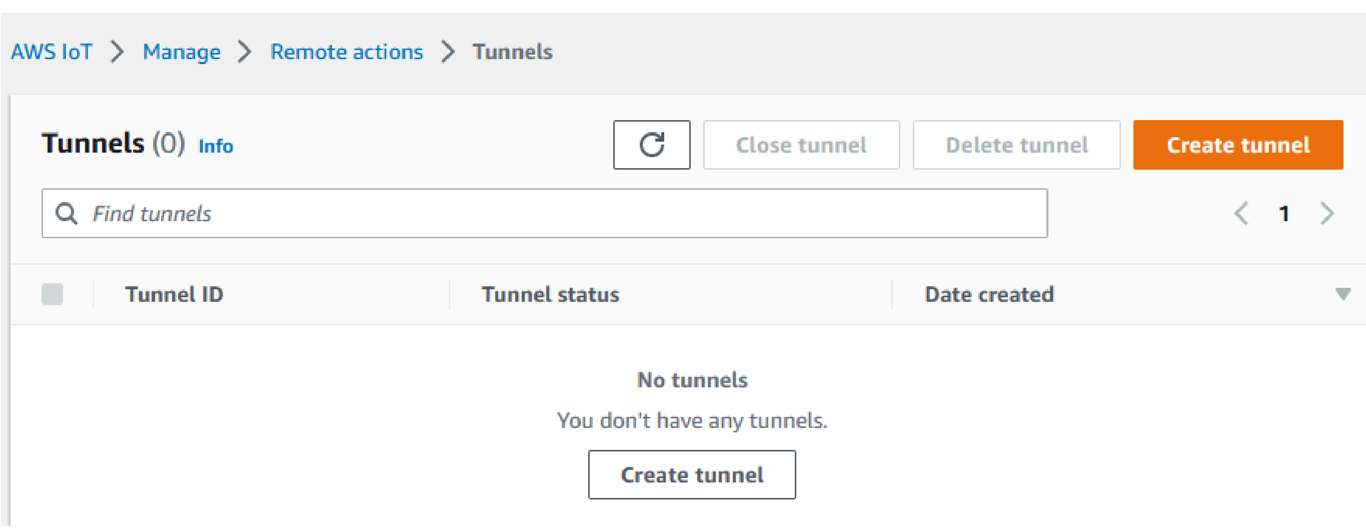
- The firewalls that the remote device is behind must allow outbound traffic on port 443. The tunnel that you create will use this port to connect to the remote device.
- You have an IoT device agent (see IoT agent snippet) running on the remote device that connects to the AWS IoT device gateway and is configured with an MQTT topic subscription. For more information, see connect a device to the AWS IoT device gateway.
- You must have an SSH daemon running on the remote device.
- You have downloaded the local proxy source code from GitHub and built it for the platform of your choice. We'll refer to the built local proxy executable file as `localproxy` in this tutorial.

Open a tunnel

You can open a secure tunnel using the AWS Management Console, the AWS IoT API Reference, or the AWS CLI. You can optionally configure a destination name but it's not required for this tutorial. If you configure the destination, secure tunneling will automatically deliver the access token to the remote device using MQTT. For more information, see Tunnel creation methods in AWS IoT console.

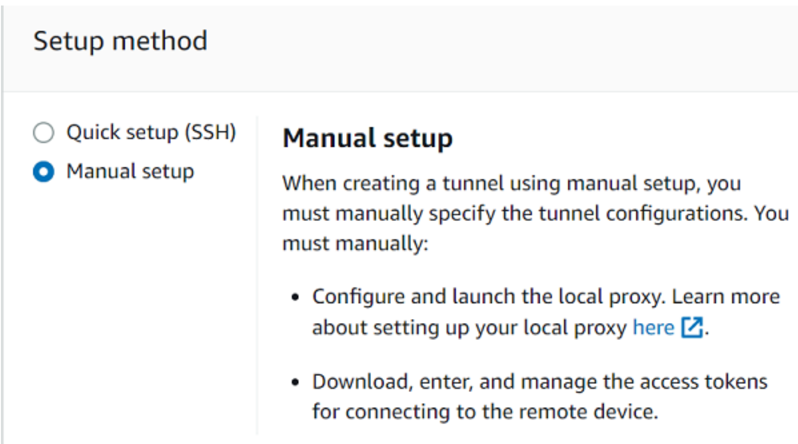
To open a tunnel in the console

1. Go to the Tunnels hub of the AWS IoT console and choose **Create tunnel**.



2. For this tutorial, choose **Manual setup** as the tunnel creation method and then choose **Next**. For information about using the **quick setup** method to create a tunnel, see Open a tunnel and use browser-based SSH to access remote device.

> **Note**
>
> If you create a secure tunnel from the details page of a thing, you can choose whether to create a new tunnel or use an existing one. For more information, see Open a tunnel for remote device and use browser-based SSH.



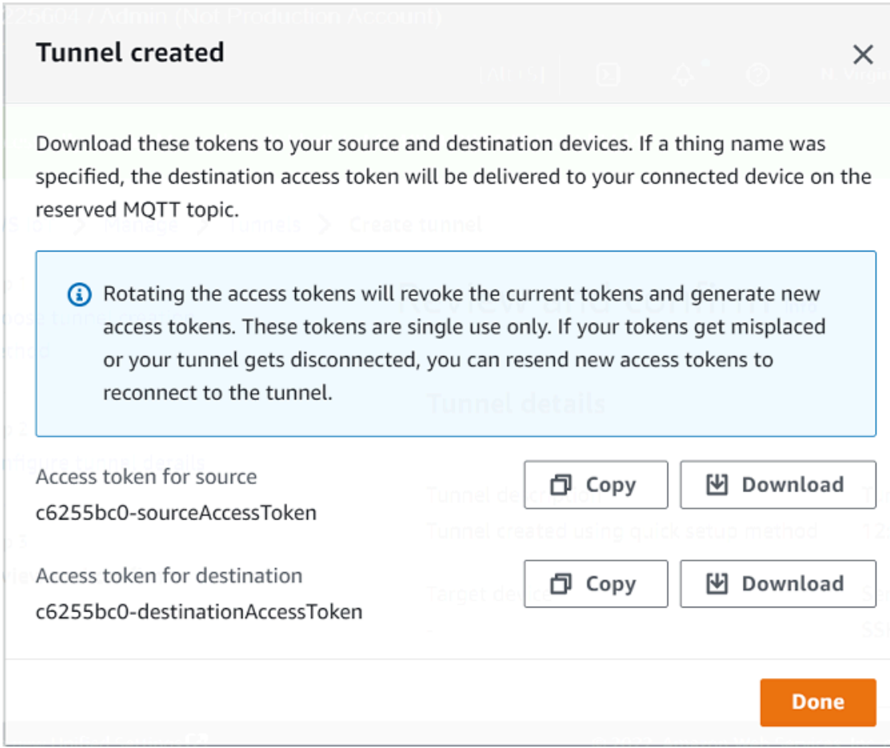
3. (Optional) Enter the configuration settings for your tunnel. You can also skip this step and proceed to the next step to create a tunnel.

   Enter a tunnel description, a tunnel timeout duration, and resource tags as key-value pairs to help you identify your resource. For this tutorial, you can skip the destination configuration.

> **Note**
>
> You won't be charged based on the duration for which you keep a tunnel open. You only incur charges when creating a new tunnel. For pricing information, see **Secure Tunneling** in AWS IoT Device Management pricing.

4. Download the client access tokens and then choose **Done**. The tokens will not be available to download after you choose **Done**.

These tokens can only be used once to connect to the tunnel. If you misplace the tokens or the tunnel gets disconnected, you can generate and send new tokens to your remote device for reconnecting to the tunnel.



To open a tunnel using the API

To open a new tunnel, you can use the OpenTunnel API operation. You can also specify additional configurations using the API, such as the tunnel duration and the destination configuration.

```
aws iotsecuretunneling open-tunnel \
    --region us-east-1 \
    --endpoint https://api.us-east-1.tunneling.iot.amazonaws.com
```

Running this command creates a new tunnel and provides you the source and destination access tokens.

```
{
    "tunnelId": "01234567-89ab-0123-4c56-789a01234bcd",
    "tunnelArn": "arn:aws:iot:us-
east-1:123456789012:tunnel/01234567-89ab-0123-4c56-789a01234bcd",
    "sourceAccessToken": "<SOURCE_ACCESS_TOKEN>",
    "destinationAccessToken": "<DESTINATION_ACCESS_TOKEN>"
```

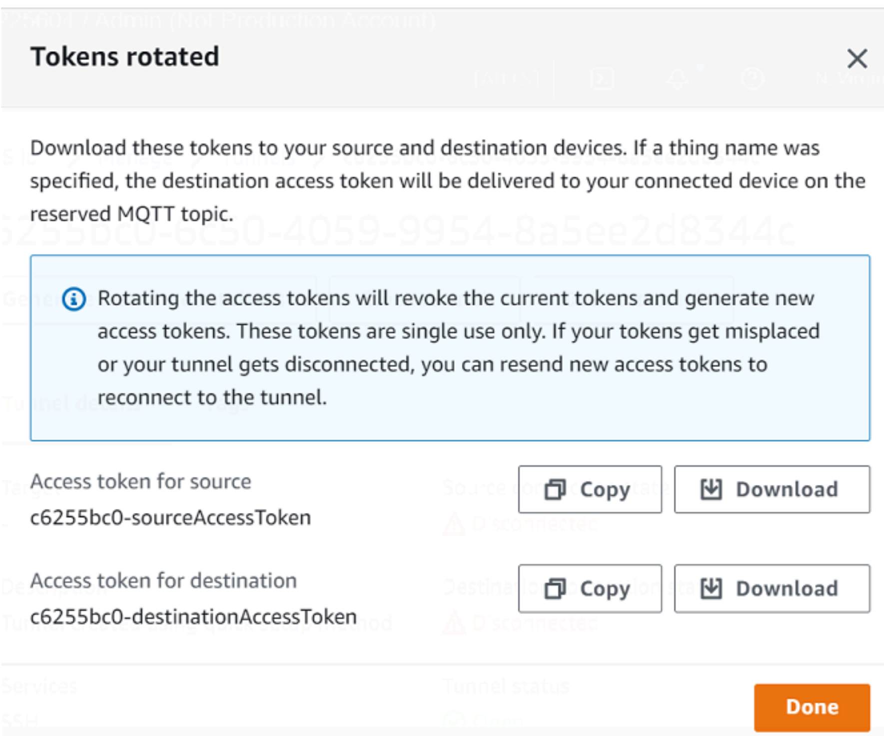
```
}
```

Resend tunnel access tokens

The tokens that you obtained when creating a tunnel can only be used once to connect to the tunnel. If you misplace the access token or the tunnel gets disconnected, you can resend new access tokens to the remote device using MQTT at no additional charge. AWS IoT secure tunneling will revoke the current tokens and return new access tokens for reconnecting to the tunnel.

To rotate the tokens from the console

1. Go to the Tunnels hub of the AWS IoT console and choose the tunnel that you created.
2. In the tunnel details page, choose **Generate new access tokens** and then choose **Next**.
3. Download the new access tokens for your tunnel and choose **Done**. These tokens can be used only once. If you misplace these tokens or the tunnel gets disconnected, you can resend new access tokens.



To rotate access tokens using the API

To rotate the tunnel access tokens, you can use the RotateTunnelAccessToken API operation to revoke the current tokens and return new access tokens for reconnecting to the tunnel.

For example, the following command rotates the access tokens for the destination device, *RemoteThing1*.

```
aws iotsecuretunneling rotate-tunnel-access-token \
    --tunnel-id <tunnel-id> \
    --client-mode DESTINATION \
    --destination-config thingName=<RemoteThing1>,services=SSH \
    --region <region>
```

Running this command generates the new access token as shown in the following example. The token is then delivered to the device using MQTT to connect to the tunnel, if the device agent is set up correctly.

```
{
    "destinationAccessToken": "destination-access-token",
    "tunnelArn": "arn:aws:iot:region:account-id:tunnel/tunnel-id"
}
```

For examples that show how and when to rotate the access tokens, see [Resolving AWS IoT secure tunneling connectivity issues by rotating client access tokens].

Configure and start the local proxy

To connect to the remote device, open a terminal on your laptop and configure and start the local proxy. The local proxy transmits data sent by the application running on the source device by using secure tunneling over a WebSocket secure connection. You can download the local proxy source from [GitHub].

After you configure the local proxy, copy the source client access token, and use it to start the local proxy in source mode. Following shows an example command to start the local proxy. In the following command, the local proxy is configured to listen for new connections on port 5555. In this command:

- `-r` specifies the AWS Region, which must be the same Region where your tunnel was created.
- `-s` specifies the port to which the proxy should connect.
- `-t` specifies the client token text.

```
./localproxy -r us-east-1 -s 5555 -t source-client-access-token
```

Running this command will start the local proxy in source mode. If you receive the following error after running the command, set up the CA path. For information, see [Secure tunneling local proxy on GitHub](#).

```
Could not perform SSL handshake with proxy server: certificate verify failed
```

The following shows a sample output of running the local proxy in `source` mode.

```
...
...

Starting proxy in source mode
Attempting to establish web socket connection with endpoint wss://
data.tunneling.iot.us-east-1.amazonaws.com:443
Resolved proxy  server IP: 10.10.0.11
Connected successfully with proxy server
Performing SSL handshake with proxy server
Successfully completed SSL handshake with proxy server
HTTP/1.1 101 Switching Protocols

...

Connection: upgrade
channel-id: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
upgrade: websocket

...

Web socket session ID: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-east-1.amazonaws.com:443
Setting up web socket pings for every 5000 milliseconds
Scheduled next read:

...

Starting web socket read loop continue reading...
Resolved bind IP: 127.0.0.1
```

```
Listening for new connection on port 5555
```

Start an SSH session

Open another terminal and use the following command to start a new SSH session by connecting to the local proxy on port 5555.

```
ssh username@localhost -p 5555
```

You might be prompted for a password for the SSH session. When you are done with the SSH session, type **exit** to close the session.

Cleaning up

- **Close tunnel**

  We recommend that you close the tunnel after you've finished using it. A tunnel can also become closed if it stayed open for longer than the specified tunnel duration. A tunnel cannot be reopened once closed. You can still duplicate a tunnel by opening the closed tunnel and then choosing **Duplicate tunnel**. Specify the tunnel duration that you want to use and then create the new tunnel.

  - To close an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to close, and then choose **Close tunnel**.
  - To close an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API operation.

    ```
    aws iotsecuretunneling close-tunnel \
        --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
    ```

- **Delete tunnel**

  You can delete a tunnel permanently from your AWS account.

  > **Warning**
  >
  > Deletion actions are permanent and can't be undone.

- To delete an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to delete, and then choose **Delete tunnel**.
- To delete an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API operation. When using the API, set the `delete` flag to `true`.

```
aws iotsecuretunneling close-tunnel \
    --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
    --delete true
```

Open a tunnel for remote device and use browser-based SSH

From the AWS IoT console, you can create a tunnel either from the **Tunnels hub** or from the details page of an IoT thing that you created. When you create a tunnel from the **Tunnels** hub, you can specify whether to create a tunnel using the quick setup or the manual setup. For an example tutorial, see Open a tunnel and start SSH session to remote device.

When you create a tunnel from the thing details page of the AWS IoT console, you can also specify whether to create a new tunnel or open an existing tunnel for that thing as illustrated in this tutorial. If you choose an existing tunnel, you can access the most recent, open tunnel that you created for this device. You can then use the command line interface within the terminal to SSH into the device.

Prerequisites

- The firewalls that the remote device is behind must allow outbound traffic on port 443. The tunnel that you create will use this port to connect to the remote device.
- You have created an IoT thing (for example, `RemoteDevice1`) in the AWS IoT registry. This thing corresponds to the representation of your remote device in the cloud. For more information, see Register a device in the AWS IoT registry.
- You have an IoT device agent (see IoT agent snippet) running on the remote device that connects to the AWS IoT device gateway and is configured with an MQTT topic subscription. For more information, see connect a device to the AWS IoT device gateway.
- You must have an SSH daemon running on the remote device.

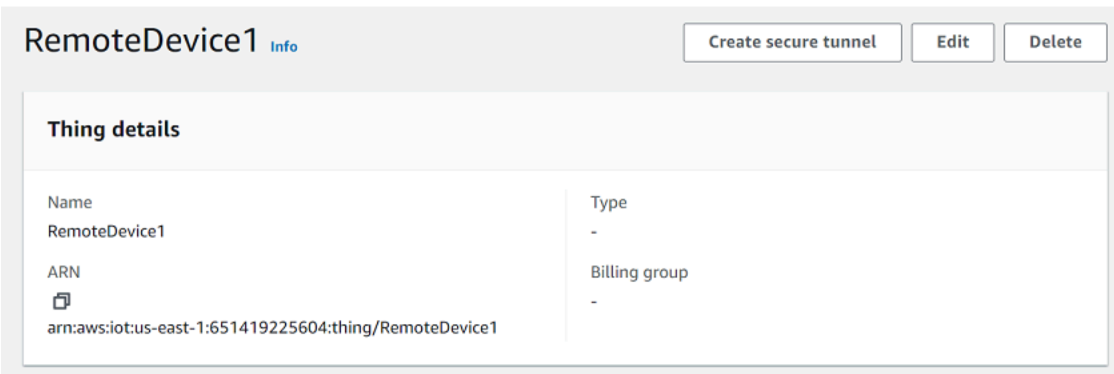
Open a new tunnel for the remote device

Say you want to open a tunnel into your remote device, `RemoteDevice1`. First, create an IoT thing with the name `RemoteDevice1` in the AWS IoT registry. You can then create a tunnel using the AWS Management Console, the AWS IoT API Reference API, or the AWS CLI.

By configuring a destination when creating a tunnel, the secure tunneling service delivers the destination client access token to the remote device over MQTT and the reserved MQTT topic (`$aws/things/RemoteDeviceA/tunnels/notify`). For more information, see Tunnel creation methods in AWS IoT console.

To create a tunnel for remote device from console

1. Choose the thing, `RemoteDevice1`, to view its details, and then choose **Create secure tunnel**.



2. Choose whether to create a new tunnel or open an existing tunnel. To create a new tunnel, choose **Create new tunnel**. You can then choose whether to use the manual setup or the quick setup method to create the tunnel. For more information, see Open a tunnel using manual setup and connect to remote device and Open a tunnel and use browser-based SSH to access remote device.

To create a tunnel for remote device using API

To open a new tunnel, you can use the OpenTunnel API operation. The following code shows an example of running this command.

```
aws iotsecuretunneling open-tunnel \
    --region us-east-1 \
    --endpoint https://api.us-east-1.tunneling.iot.amazonaws.com
    --cli-input-json file://input.json
```

Following shows the contents for the `input.json` file. You can use the `destinationConfig` parameter to specify the name of the destination device (for example, *`RemoteDevice1`*) and the service that you want to use to access the destination device, such as *SSH*. Optionally, you can also specify additional parameters such as tunnel description and tags.

Contents of input.json

```
{
   "description": "Tunnel to remote device1",
   "destinationConfig": {
      "services": [ "SSH" ],
      "thingName": "RemoteDevice1"
   }
}
```

Running this command creates a new tunnel and provides you the source and destination access tokens.

```
{
    "tunnelId": "01234567-89ab-0123-4c56-789a01234bcd",
    "tunnelArn": "arn:aws:iot:us-
east-1:123456789012:tunnel/01234567-89ab-0123-4c56-789a01234bcd",
    "sourceAccessToken": "<SOURCE_ACCESS_TOKEN>",
    "destinationAccessToken": "<DESTINATION_ACCESS_TOKEN>"
}
```

Open an existing tunnel and use browser-based SSH

Say you created the tunnel for your remote device, `RemoteDevice1`, using the manual setup method or using the AWS IoT API Reference API. You can then open the existing tunnel for the device and choose **Quick setup** to use the browser-based SSH feature. The configurations of an existing tunnel can't be edited so you can't use the manual setup method.

To use the browser-based SSH feature, you won't have to download the source access token or configure the local proxy. A web-based local proxy will be automatically configured for you so you can start interacting with your remote device.

To use the quick setup method and browser-based SSH

1. Go to the details page of the thing that you created, `RemoteDevice1`, and **Create secure tunnel**.

2. Choose **Use existing tunnel** to open the most recent, open tunnel that you created for the remote device. The tunnel configurations can't be edited so you can't use the manual setup method for the tunnel. To use the quick setup method, choose **Quick setup**.

3. Proceed to review and confirm the tunnel configuration details and create the tunnel. The tunnel configurations can't be edited.

   When you create the tunnel, secure tunneling will use the RotateTunnelAccessToken API operation to revoke the original access tokens and generate new access tokens. If your remote device uses MQTT, these tokens will be automatically delivered to the remote device on the MQTT topic that it's subscribed to. You can also choose to download these tokens manually to your source device.

After you've created the tunnel, you can use the browser-based SSH to interact with the remote device directly from the console using the in-context command-line interface. To use this command- line interface, choose the tunnel for the thing that you created, and in the details page, expand the **Command-line interface** section. As the local proxy has already been configured for you, you can start entering commands to quickly get started with accessing and interacting with your remote device, `RemoteDevice1`.

For more information about the quick setup method and using the browser-based SSH, see Open a tunnel and use browser-based SSH to access remote device.

Cleaning up

- **Close tunnel**

  We recommend that you close the tunnel after you've finished using it. A tunnel can also become closed if it stayed open for longer than the specified tunnel duration. A tunnel cannot be reopened once closed. You can still duplicate a tunnel by opening the closed tunnel and then choosing **Duplicate tunnel**. Specify the tunnel duration that you want to use and then create the new tunnel.

  - To close an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to close, and then choose **Close tunnel**.
  - To close an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API operation.

```
aws iotsecuretunneling close-tunnel \
```

```
    --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
```

- **Delete tunnel**

  You can delete a tunnel permanently from your AWS account.

  > **Warning**
  >
  > Deletion actions are permanent and can't be undone.

  - To delete an individual tunnel or multiple tunnels from the AWS IoT console, go to the Tunnels hub, choose the tunnels that you want to delete, and then choose **Delete tunnel**.
  - To delete an individual tunnel or multiple tunnels using the AWS IoT API Reference API, use the CloseTunnel API operation. When using the API, set the `delete` flag to `true`.

  ```
  aws iotsecuretunneling close-tunnel \
      --tunnel-id "01234567-89ab-0123-4c56-789a01234bcd"
      --delete true
  ```

Local proxy

The local proxy transmits data sent by the application running on the source device by using secure tunneling over a WebSocket secure connection. You can download the local proxy source from GitHub.

The local proxy can run in two modes: `source` or `destination`. In source mode, the local proxy runs on the same device or network as the client application that initiates the TCP connection. In destination mode, the local proxy runs on the remote device, along with the destination application. A single tunnel can support up to three data streams at a time by using tunnel multiplexing. For each data stream, secure tunneling uses multiple TCP connections, which reduces the potential for a time out. For more information, see Multiplex data streams and using simultaneous TCP connections in a secure tunnel.

How to use the local proxy

You can run the local proxy on the source and destination devices to transmit data to the secure tunneling endpoints. If your devices are in a network that use a web proxy, the web proxy can intercept the connections before forwarding them to the internet. In this case, you'll need to configure your local proxy to use the web proxy. For more information, see Configure local proxy for devices that use web proxy.

Local proxy workflow

The following steps show how the local proxy is run on the source and destination devices.

1. **Connect local proxy to secure tunneling**

   First, local proxy must establish a connection to secure tunneling. When you start the local proxy, use the following arguments:

   - The `-r` argument to specify the AWS Region in which the tunnel is opened.
   - The `-t` argument to pass either the source or destination client access token returned from the `OpenTunnel`.

   > **Note**
   >
   > Two local proxies using the same client access token value cannot be connected at the same time.

2. **Perform source or destination actions**

   After the WebSocket connection is established, the local proxy performs either source mode or destination mode actions, depending on its configuration.

   By default, the local proxy attempts to reconnect to secure tunneling if any input/output (I/O) errors occur or if the WebSocket connection is closed unexpectedly. This causes the TCP connection to close. If any TCP socket errors occur, the local proxy sends a message through the tunnel to notify the other side to close its TCP connection. By default, the local proxy always uses SSL communication.

3. **Stop the local proxy**

   After you use the tunnel, it is safe to stop the local proxy process. We recommend that you explicitly close the tunnel by calling `CloseTunnel`. Active tunnel clients might not be closed right after calling `CloseTunnel`.

For more information about how to use the AWS Management Console to open a tunnel and start an SSH session, see Open a tunnel and start SSH session to remote device.

Local proxy best practices

When running the local proxy, follow these best practices:

- Avoid the use of the `-t` local proxy argument to pass in an access token. We recommend that you use the `AWSIOT_TUNNEL_ACCESS_TOKEN` environment variable to set the access token for the local proxy.
- Run the local proxy executable with least privileges in the operating system or environment.
  - Avoid running the local proxy as an administrator on Windows.
  - Avoid running the local proxy as root on Linux and macOS.
- Consider running the local proxy on separate hosts, containers, sandboxes, chroot jail, or a virtualized environment.
- Build the local proxy with relevant security flags, depending on your toolchain.
- On devices with multiple network interfaces, use the `-b` argument to bind the TCP socket to the network interface used to communicate with the destination application.

Example command and output

The following shows an example of a command that you run and the corresponding output. The example shows how the local proxy can be configured in both `source` and `destination` modes. The local proxy upgrades the HTTPS protocol to WebSockets to establish a long-lived connection and then starts transmitting data through the connection to the secure tunneling device endpoints.

Before you run these commands:

You must have opened a tunnel and obtained the client access tokens for the source and destination. You must have also built the local proxy as described previously. To build the local proxy, open the local proxy source code in the GitHub repository and follow the instructions for building and installing the local proxy.

> **Note**
>
> The following commands used in the examples use the `verbosity` flag to illustrate an overview of the different steps described previously after you run the local proxy. We recommend that you use this flag only for testing purposes.

Running local proxy in source mode

The following commands display how to run the local proxy in source mode.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure and start the local proxy on your source.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
./localproxy -s 5555 -v 5 -r us-west-2
```

Where:

- `-s` is the source listen port, which starts the local proxy in source mode.
- `-v` is the verbosity of the output, which can be a value between zero and six.
- `-r` is the endpoint region where the tunnel is opened.

For more information about the parameters, see Options set using command line arguments.

Windows

In Windows, you configure the local proxy similar to how you do for Linux or macOS, but how you define the environment variables is different from the other platforms. Run the following commands in the `cmd` window to configure and start the local proxy on your source.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
.\localproxy -s 5555 -v 5 -r us-west-2
```

Where:

- `-s` is the source listen port, which starts the local proxy in source mode.
- `-v` is the verbosity of the output, which can be a value between zero and six.

- `-r` is the endpoint region where the tunnel is opened.

For more information about the parameters, see Options set using command line arguments.

> **Note**
>
> When using the latest version of local proxy in source mode, you must include the AWS CLI parameter `--destination-client-type V1` on the source device for backward compatibility. This applies when connecting to any of these destination modes:
>
> - AWS IoT Device Client
> - AWS IoT Secure Tunneling Component or AWS IoT Greengrass Version 2 Secure Tunneling Component
> - Any AWS IoT Secure Tunneling demo code written before 2022
> - 1.X versions of the local proxy
>
> This parameter ensures proper communication between the updated source proxy and older destination clients. For more information about local proxy versions, see AWS IoT Secure Tunneling on *GitHub*.

The following is a sample output of running the local proxy in `source` mode.

```
...
...

Starting proxy in source mode
Attempting to establish web socket connection with endpoint wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Resolved proxy  server IP: 10.10.0.11
Connected successfully with proxy server
Performing SSL handshake with proxy server
Successfully completed SSL handshake with proxy server
HTTP/1.1 101 Switching Protocols

...

Connection: upgrade
```

```
channel-id: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
upgrade: websocket

...

Web socket session ID: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Setting up web socket pings for every 5000 milliseconds
Scheduled next read:

...

Starting web socket read loop continue reading...
Resolved bind IP: 127.0.0.1
Listening for new connection on port 5555
```

Running local proxy in destination mode

The following commands display how to run the local proxy in destination mode.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure and start the local proxy on your destination.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
./localproxy -d 22 -v 5 -r us-west-2
```

Where:

- `-d` is the destination application which starts the local proxy in destination mode.
- `-v` is the verbosity of the output, which can be a value between zero and six.
- `-r` is the endpoint region where the tunnel is opened.

For more information about the parameters, see [Options set using command line arguments](#).

Windows

In Windows, you configure the local proxy similar to how you do for Linux or macOS, but how you define the environment variables is different from the other platforms. Run the following commands in the `cmd` window to configure and start the local proxy on your destination.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
.\localproxy -d 22 -v 5 -r us-west-2
```

Where:

- `-d` is the destination application which starts the local proxy in destination mode.
- `-v` is the verbosity of the output, which can be a value between zero and six.
- `-r` is the endpoint region where the tunnel is opened.

For more information about the parameters, see Options set using command line arguments.

> **Note**
>
> When using the latest version of local proxy in destination mode, you must include the AWS CLI parameter `--destination-client-type V1` on the destination device for backward compatibility. This applies when connecting to either of these source modes:
>
> - Browser-based Secure Tunneling from the AWS Console.
> - 1.X versions of the local proxy
>
> This parameter ensures proper communication between the updated destination proxy and older source clients. For more information about local proxy versions, see AWS IoT Secure Tunneling on *GitHub*.

The following is a sample output of running the local proxy in `destination` mode.

```
...
...

Starting proxy in destination mode
```

```
Attempting to establish web socket connection with endpoint wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Resolved proxy  server IP: 10.10.0.11
Connected successfully with proxy server
Performing SSL handshake with proxy server
Successfully completed SSL handshake with proxy server
HTTP/1.1 101 Switching Protocols

...

Connection: upgrade
channel-id: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
upgrade: websocket

...

Web socket session ID: 01234567890abc23-00001234-0005678a-b1234c5de677a001-2bc3d456
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Setting up web socket pings for every 5000 milliseconds
Scheduled next read:

...

Starting web socket read loop continue reading...
```

Configure local proxy for devices that use web proxy

You can use local proxy on AWS IoT devices to communicate with AWS IoT secure tunneling APIs. The local proxy transmits data sent by the device application using secure tunneling over a WebSocket secure connection. The local proxy can work in `source` or `destination` mode. In `source` mode, it runs on the same device or network that initiates the TCP connection. In `destination` mode, the local proxy runs on the remote device, along with the destination application. For more information, see Local proxy.

The local proxy needs to connect directly to the internet to use AWS IoT secure tunneling. For a long-lived TCP connection with secure tunneling, the local proxy upgrades the HTTPS request to establish a WebSockets connection to one of the secure tunneling device connection endpoints.

If your devices are in a network that uses a web proxy, the web proxy can intercept the connections before forwarding them to the internet. To establish a long-lived connection to the secure tunneling device connection endpoints, configure your local proxy to use the web proxy as described in the websocket specification.

> **Note**
>
> The AWS IoT Device Client doesn't support devices that use a web proxy. To work with the web proxy, you'll need to use a local proxy and configure it to work with a web proxy as described below.

The following steps show how the local proxy works with a web proxy.

1. The local proxy sends an HTTP `CONNECT` request to the web proxy that contains the remote address of the secure tunneling service, along with the web proxy authentication information.
2. The web proxy will then create a long-lived connection to the remote secure tunneling endpoints.
3. The TCP connection is established and the local proxy will now work in both source and destination modes for data transmission.

To complete this procedure, perform the following steps.

- Build the local proxy
- Configure your web proxy
- Configure and start the local proxy

Build the local proxy

Open the local proxy source code in the GitHub repository and follow the instructions for building and installing the local proxy.

Configure your web proxy

The local proxy relies on the HTTP tunneling mechanism described by the HTTP/1.1 specification. To comply with the specifications, your web proxy must allow devices to use the `CONNECT` method.

How you configure your web proxy depends on the web proxy you're using and the web proxy version. To make sure you configure the web proxy correctly, check your web proxy's documentation.

To configure your web proxy, first identify your web proxy URL and confirm whether your web proxy supports HTTP tunneling. The web proxy URL will be used later when you configure and start the local proxy.

1. **Identify your web proxy URL**

   Your web proxy URL will be in the following format.

   ```
   protocol://web_proxy_host_domain:web_proxy_port
   ```

   AWS IoT secure tunneling supports only basic authentication for web proxy. To use basic authentication, you must specify the **username** and **password** as part of the web proxy URL. The web proxy URL will be in the following format.

   ```
   protocol://username:password@web_proxy_host_domain:web_proxy_port
   ```

   - *protocol* can be `http` or `https`. We recommend that you use `https`.
   - *web_proxy_host_domain* is the IP address of your web proxy or a DNS name that resolves to the IP address of your web proxy.
   - *web_proxy_port* is the port on which the web proxy is listening.
   - The web proxy uses this **username** and **password** to authenticate the request.

2. **Test your web proxy URL**

   To confirm whether your web proxy supports TCP tunneling, use a `curl` command and make sure that you get a `2xx` or a `3xx` response.

   For example, if your web proxy URL is `https://server.com:1235`, use a `proxy-insecure` flag with the `curl` command because the web proxy might rely on a self-signed certificate.

   ```
   export HTTPS_PROXY=https://server.com:1235
   curl -I https://aws.amazon.com --proxy-insecure
   ```

   If your web proxy URL has a `http` port (for example, `http://server.com:1234`), you don't have to use the `proxy-insecure` flag.

   ```
   export HTTPS_PROXY=http://server.com:1234
   curl -I https://aws.amazon.com
   ```

Configure and start the local proxy

To configure the local proxy to use a web proxy, you must configure the `HTTPS_PROXY` environment variable with either the DNS domain names or the IP addresses and port numbers that your web proxy uses.

After you've configured the local proxy, you can use the local proxy as explained in this README document.

> **Note**
>
> Your environment variable declaration is case sensitive. We recommend that you define each variable once using either all uppercase or all lowercase letters. The following examples show the environment variable declared in uppercase letters. If the same variable is specified using both uppercase and lowercase letters, the variable specified using lowercase letters takes precedence.

The following commands show how to configure the local proxy that is running on your destination to use the web proxy and start the local proxy.

- `AWSIOT_TUNNEL_ACCESS_TOKEN`: This variable holds the client access token (CAT) for the destination.
- `HTTPS_PROXY`: This variable holds the web proxy URL or the IP address for configuring the local proxy.

The commands shown in the following examples depend on the operating system that you use and whether the web proxy is listening on an HTTP or an HTTPS port.

Web proxy listening on an HTTP port

If your web proxy is listening on an HTTP port, you can provide the web proxy URL or IP address for the `HTTPS_PROXY` variable.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure and start the local proxy on your destination to use a web proxy listening to an HTTP port.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
```

```
export HTTPS_PROXY=http:proxy.example.com:1234
./localproxy -r us-east-1 -d 22
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the HTTPS_PROXY variable.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://username:password@proxy.example.com:1234
./localproxy -r us-east-1 -d 22
```

Windows

In Windows, you configure the local proxy similar to how you do for Linux or macOS, but how you define the environment variables is different from the other platforms. Run the following commands in the cmd window to configure and start the local proxy on your destination to use a web proxy listening to an HTTP port.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://proxy.example.com:1234
.\localproxy -r us-east-1 -d 22
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the HTTPS_PROXY variable.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://username:password@10.15.20.25:1234
.\localproxy -r us-east-1 -d 22
```

Web proxy listening on an HTTPS port

Run the following commands if your web proxy is listening on an HTTPS port.

Note

If you're using a self-signed certificate for the web proxy or if you're running the local proxy on an OS that doesn't have native OpenSSL support and default configurations, you'll have to set up your web proxy certificates as described in the Certificate setup section in the GitHub repository.

The following commands will look similar to how you configured your web proxy for an HTTP proxy, with the exception that you'll also specify the path to the certificate files that you installed as described previously.

Linux/macOS

In Linux or macOS, run the following commands in the terminal to configure the local proxy running on your destination to use a web proxy listening to an HTTPS port.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http:proxy.example.com:1234
./localproxy -r us-east-1 -d 22 -c /path/to/certs
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the HTTPS_PROXY variable.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http://username:password@proxy.example.com:1234
./localproxy -r us-east-1 -d 22 -c /path/to/certs
```

Windows

In Windows, run the following commands in the cmd window to configure and start the local proxy running on your destination to use a web proxy listening to an HTTP port.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://proxy.example.com:1234
.\localproxy -r us-east-1 -d 22 -c \path\to\certs
```

If you have to authenticate with the proxy, you must specify a **username** and **password** as part of the HTTPS_PROXY variable.

```
set AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
set HTTPS_PROXY=http://username:password@10.15.20.25:1234
.\localproxy -r us-east-1 -d 22 -c \path\to\certs
```

Example command and output

The following shows an example of a command that you run on a Linux OS and the corresponding output. The example shows a web proxy that's listening on an HTTP port and how the local proxy can be configured to use the web proxy in both `source` and `destination` modes. Before you can run these commands, you must have already opened a tunnel and obtained the client access tokens for the source and destination. You must have also built the local proxy and configured your web proxy as described previously.

Here's an overview of the steps after you start the local proxy. The local proxy:

- Identifies the web proxy URL so that it can use the URL to connect to the proxy server.
- Establishes a TCP connection with the web proxy.
- Sends an HTTP `CONNECT` request to the web proxy and waits for the `HTTP/1.1 200` response, which indicates that connection has been established.
- Upgrades the HTTPS protocol to WebSockets to establish a long-lived connection.
- Starts transmitting data through the connection to the secure tunneling device endpoints.

> **Note**
>
> The following commands used in the examples use the `verbosity` flag to illustrate an overview of the different steps described previously after you run the local proxy. We recommend that you use this flag only for testing purposes.

Running local proxy in source mode

The following commands show how to run the local proxy in source mode.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http:username:password@10.15.10.25:1234
./localproxy -s 5555 -v 5 -r us-west-2
```

The following shows a sample output of running the local proxy in `source` mode.

```
...

Parsed basic auth credentials for the URL
Found Web proxy information in the environment variables, will use it to connect via
 the proxy.

...
```

```
Starting proxy in source mode
Attempting to establish web socket connection with endpoint wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Resolved Web proxy IP: 10.10.0.11
Connected successfully with Web Proxy
Successfully sent HTTP CONNECT to the Web proxy
Full response from the Web proxy:
HTTP/1.1 200 Connection established
TCP tunnel established successfully
Connected successfully with proxy server
Successfully completed SSL handshake with proxy server
Web socket session ID: 0a109afffee745f5-00001341-000b8138-cc6c878d80e8adb0-f186064b
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Seting up web socket pings for every 5000 milliseconds
Scheduled next read:

...

Starting web socket read loop continue reading...
Resolved bind IP: 127.0.0.1
Listening for new connection on port 5555
```

Running local proxy in destination mode

The following commands show how to run the local proxy in destination mode.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=${access_token}
export HTTPS_PROXY=http:username:password@10.15.10.25:1234
./localproxy -d 22 -v 5 -r us-west-2
```

The following shows a sample output of running the local proxy in `destination` mode.

```
...

Parsed basic auth credentials for the URL
Found Web proxy information in the environment variables, will use it to connect via
 the proxy.

...
```

```
Starting proxy in destination mode
Attempting to establish web socket connection with endpoint wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Resolved Web proxy IP: 10.10.0.1
Connected successfully with Web Proxy
Successfully sent HTTP CONNECT to the Web proxy
Full response from the Web proxy:
HTTP/1.1 200 Connection established
TCP tunnel established successfully
Connected successfully with proxy server
Successfully completed SSL handshake with proxy server
Web socket session ID: 06717bfffed3fd05-00001355-000b8315-da3109a85da804dd-24c3d10d
Web socket subprotocol selected: aws.iot.securetunneling-2.0
Successfully established websocket connection with proxy server: wss://
data.tunneling.iot.us-west-2.amazonaws.com:443
Seting up web socket pings for every 5000 milliseconds
Scheduled next read:

...

Starting web socket read loop continue reading...
```

Multiplex data streams and using simultaneous TCP connections in a secure tunnel

You can use multiple data streams per tunnel by using the secure tunneling multiplexing feature. With multiplexing, you can troubleshoot devices using multiple data streams. You can also reduce your operational load by eliminating the need to build, deploy, and start multiple local proxies or open multiple tunnels to the same device. For example, multiplexing can be used in case of a web browser that requires sending multiple HTTP and SSH data streams.

For each data stream, AWS IoT secure tunneling supports simultaneous TCP connections. Using simultaneous connections reduces the potential for a time-out in case of multiple requests from the client. For example, it can reduce the loading time when remotely accessing a web server that's local to the destination device.

The following sections explain more about multiplexing and using simultaneous TCP connections, and their different use cases.

Topics

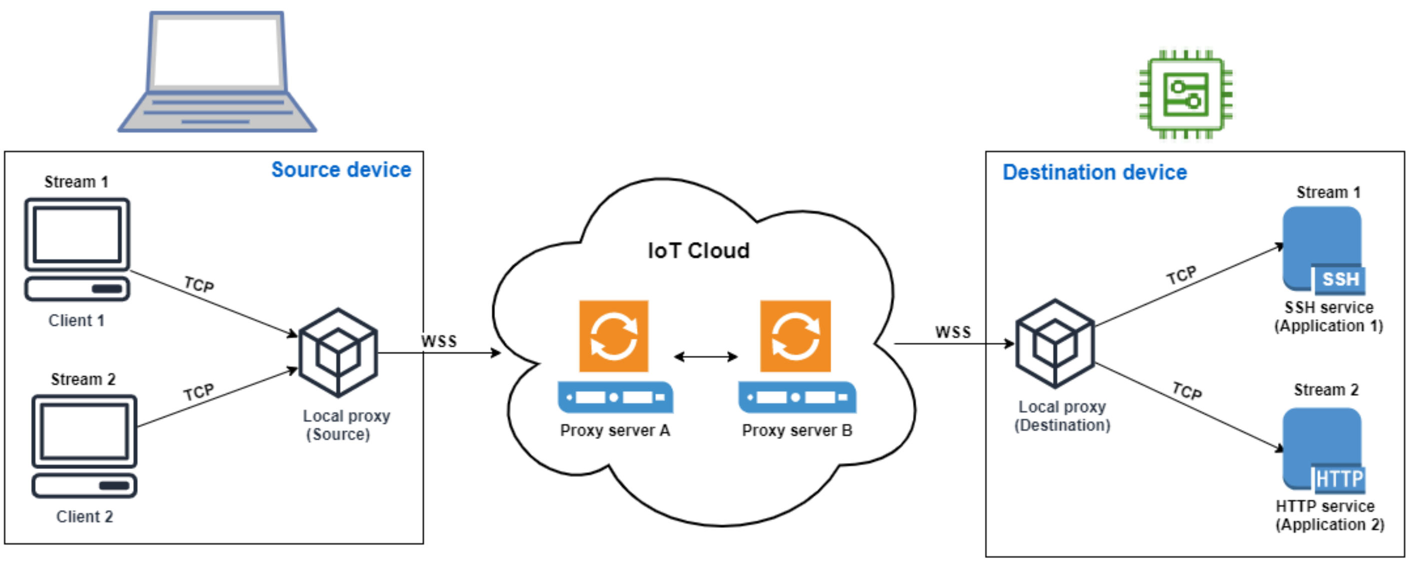
- Multiplexing multiple data streams in a secure tunnel
- Using simultaneous TCP connections in a secure tunnel

Multiplexing multiple data streams in a secure tunnel

You can use the multiplexing feature for devices that use multiple connections or ports. Multiplexing can also be used when you require multiple connections to a remote device to troubleshoot any issues. For example, it can be used in case of a web browser that requires sending multiple HTTP and SSH data streams. The application data from both streams are sent to the device concurrently over the multiplexed tunnel.

Example use case

Say you need to connect to an on-device web application to change some networking parameters, while simultaneously issuing shell commands through the terminal to verify that the device is working properly with the new networking parameters. In this scenario, you may need to connect to the device through both HTTP and SSH and transfer two parallel data streams to concurrently access the web application and terminal. With the multiplexing feature, these two independent streams can be transferred over the same tunnel at the same time.



How to set up a multiplexed tunnel

The following procedure walks you through how to set up a multiplexed tunnel for troubleshooting devices using applications that require connections to multiple ports. You will set up one tunnel with two multiplexed streams: one HTTP stream and one SSH stream.

1. **(Optional) Create configuration files**

   You can optionally configure the source and destination device with configuration files. Use configuration files if your port mappings are likely to change frequently. You can skip this step if you prefer to specify the port mapping explicitly using the CLI, or if you don't need to start the local proxy on designated listening ports. For more information about how to use configuration files, see [Options set via --config] in GitHub.

   1. On your source device, in the folder where your local proxy will run, create a configuration folder called `Config`. Inside this folder, create a file called `SSHSource.ini` with the following content:

      ```
      HTTP1 = 5555
      SSH1 = 3333
      ```

   2. On your destination device, in the folder where your local proxy will run, create a configuration folder called `Config`. Inside this folder, create a file called `SSHDestination.ini` with the following content:

      ```
      HTTP1 = 80
      SSH1 = 22
      ```

2. **Open a tunnel**

   Open a tunnel using the `OpenTunnel` API operation or the `open-tunnel` CLI command. Configure the destination by specifying `SSH1` and `HTTP1` as the services and the name of the AWS IoT thing that corresponds to your remote device. Your SSH and HTTP applications are running on this remote device. You must have already created the IoT thing in the AWS IoT registry. For more information, see Managing things with the registry.

   ```
   aws iotsecuretunneling open-tunnel \
    --destination-config thingName=RemoteDevice1,services=HTTP1,SSH1
   ```

   Running this command generates the source and destination access tokens which you'll use to run the local proxy.

   ```
   {
    "tunnelId": "b2de92a3-b8ff-46c0-b0f2-afa28b00cecd",
    "tunnelArn": "arn:aws:iot:us-west-2:431600097591:tunnel/b2de92a3-b8ff-46c0-b0f2-afa28b00cecd",
   ```

```
 "sourceAccessToken": source_client_access_token,
 "destinationAccessToken": destination_client_access_token
}
```

3. **Configure and start the local proxy**

   Before you can run the local proxy, either set up the AWS IoT Device Client, or download the local proxy source code from GitHub and build it for the platform of your choice. You can then start the destination and the source local proxy to connect to the secure tunnel. For more information about configuring and using the local proxy, see How to use the local proxy.

   > **Note**
   >
   > On your source device, if you don't use any configuration files or specify the port mapping using the CLI, you can still use the same command to run the local proxy. The local proxy in source mode will automatically pick up available ports to use and the mappings for you.

   Start local proxy using configuration files

   Run the following commands to run the local proxy in the source and destination modes using configuration files.

   ```
   // ----------------- Start the destination local proxy -----------------------
   ./localproxy -r us-east-1 -m dst -t destination_client_access_token

   // ----------------- Start the source local proxy ----------------------------
   // You also run the same command below if you want the local proxy to
   // choose the mappings for you instead of using configuration files.
   ./localproxy -r us-east-1 -m src -t source_client_access_token
   ```

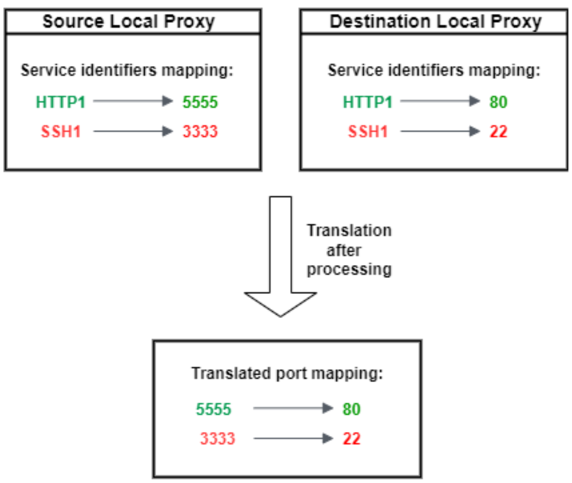
   Start local proxy using CLI port mapping

   Run the following commands to run the local proxy in the source and destination modes by specifying the port mappings explicitly using the CLI.

   ```
   // ----------------- Start the destination local proxy
    -----------------------------------
   ./localproxy -r us-east-1 -d HTTP1=80,SSH1=22 -t destination_client_access_token
   ```

```
// ---------------- Start the source local proxy
 ----------------------------------------
./localproxy -r us-east-1 -s HTTP1=5555,SSH1=33 -t source_client_access_token
```

The application data from SSH and HTTP connection can now be transferred concurrently over the multiplexed tunnel. As seen in the map below, the service identifier acts as a readable format to translate the port mapping between the source and destination device. With this configuration, secure tunneling forwards any incoming HTTP traffic from port *5555* on the source device to port *80* on the destination device, and any incoming SSH traffic from port *3333* to port *22* on the destination device.



Using simultaneous TCP connections in a secure tunnel

AWS IoT secure tunneling supports more than one TCP connection simultaneously for each data stream. You can use this capability when you require simultaneous connections to a remote device. Using simultaneous TCP connections reduces the potential for a time-out in case of multiple requests from the client. For example, when accessing a web server that has multiple components running on it, simultaneous TCP connections can reduce the time it takes to load the site.

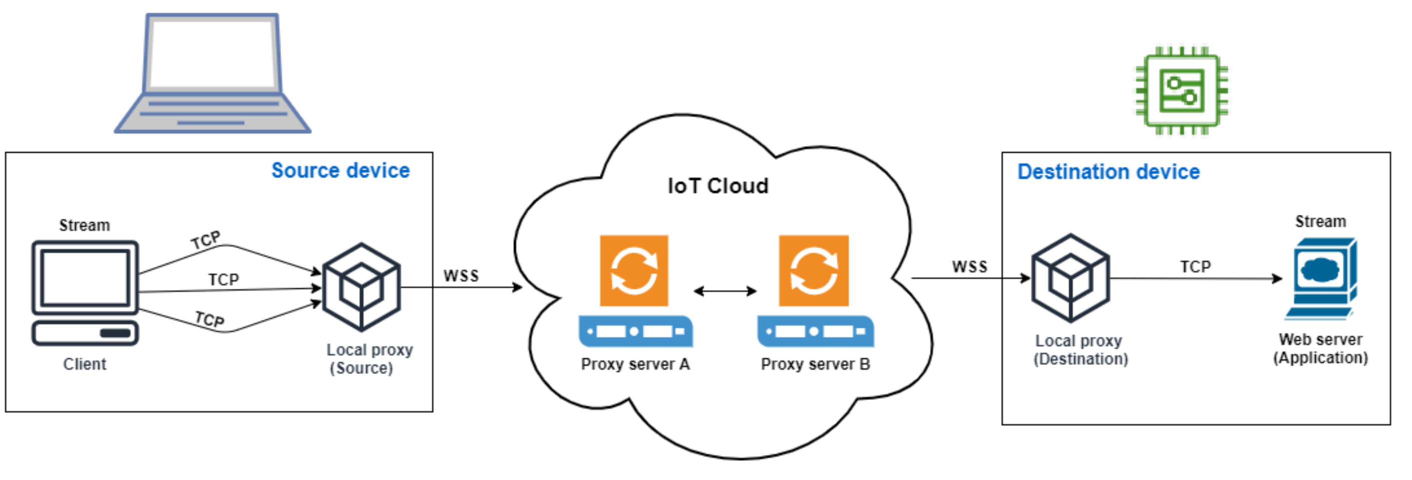
Note

Simultaneous TCP connections have a bandwidth limit of 800 Kilobytes per second for each AWS account. AWS IoT secure tunneling can configure this limit for you depending on the number of incoming requests.

Example use case

Say you need to remotely access a web server that's local to the destination device and has multiple components running on it. With a single TCP connection, while trying to access the web server, sequential loading can increase the amount of time it takes to load the resources on the site. The simultaneous TCP connections can reduce the loading time by meeting the resource requirements of the site, thereby reducing the access time. The following diagram shows how simultaneous TCP connections are supported for the data stream to the web server application running on the remote device.

> **Note**
>
> If you want to access multiple applications running on the remote device using the tunnel, you can use tunnel multiplexing. For more information, see [Multiplexing multiple data streams in a secure tunnel](#).



How to use simultaneous TCP connections

The following procedure walks you through how to use simultaneous TCP connections for accessing the web browser on the remote device. When there are multiple requests from the client, AWS IoT secure tunneling automatically sets up simultaneous TCP connections to handle the requests, thereby reducing the loading time.

1. **Open a tunnel**

   Open a tunnel using the `OpenTunnel` API operation or the `open-tunnel` CLI command. Configure the destination by specifying HTTP as the service and the name of the AWS IoT thing that corresponds to your remote device. Your web server application is running on this remote device. You must have already created the IoT thing in the AWS IoT registry. For more information, see Managing things with the registry.

```
aws iotsecuretunneling open-tunnel \
 --destination-config thingName=RemoteDevice1,services=HTTP
```

   Running this command generates the source and destination access tokens which you'll use to run the local proxy.

```
{
 "tunnelId": "b2de92a3-b8ff-46c0-b0f2-afa28b00cecd",
 "tunnelArn": "arn:aws:iot:us-west-2:431600097591:tunnel/b2de92a3-b8ff-46c0-b0f2-
afa28b00cecd",
 "sourceAccessToken": source_client_access_token,
 "destinationAccessToken": destination_client_access_token
}
```

2. **Configure and start the local proxy**

   Before you can run the local proxy, download the local proxy source code from GitHub and build it for the platform of your choice. You can then start the destination and the source local proxy to connect to the secure tunnel and start using the remote web server application.

   Note

   For AWS IoT secure tunneling to use simultaneous TCP connections, you must upgrade to the latest version of the local proxy. This feature is not available if you configure the local proxy using the AWS IoT Device Client.

```
// Start the destination local proxy
./localproxy -r us-east-1 -d HTTP=80 -t destination_client_access_token

// Start the source local proxy
```

```
./localproxy -r us-east-1 -s HTTP=5555 -t source_client_access_token
```

For more information about configuring and using the local proxy, see How to use the local proxy.

You can now use the tunnel to access the web server application. AWS IoT secure tunneling will automatically set up and handle the simultaneous TCP connections when there are multiple requests from the client.

Configuring a remote device and using IoT agent

The IoT agent is used to receive the MQTT message that includes the client access token and start a local proxy on the remote device. You must install and run the IoT agent on the remote device if you want secure tunneling to deliver the client access token using MQTT. The IoT agent must subscribe to the following reserved IoT MQTT topic:

> **Note**
>
> If you want to deliver the destination client access token to the remote device through methods other than subscribing to the reserved MQTT topic, you might need a destination client access token (CAT) listener and a local proxy. The CAT listener must work with your chosen client access token delivery mechanism and be able to start a local proxy in destination mode.

IoT agent snippet

The IoT agent must subscribe to the following reserved IoT MQTT topic so that it can receive the MQTT message and start the local proxy:

`$aws/things/thing-name/tunnels/notify`

Where `thing-name` is the name of AWS IoT thing associated with the remote device.

The following is an example MQTT message payload:

```
{
    "clientAccessToken": "destination-client-access-token",
```

```
    "clientMode": "destination",
    "region": "aws-region",
    "services": ["destination-service"]
}
```

After it receives an MQTT message, the IoT agent must start a local proxy on the remote device with the appropriate parameters.

The following Java code demonstrates how to use the AWS IoT Device SDK and ProcessBuilder from the Java library to build a simple IoT agent to work with secure tunneling.

```
// Find the IoT device endpoint for your AWS account
final String endpoint = iotClient.describeEndpoint(new
 DescribeEndpointRequest().withEndpointType("iot:Data-ATS")).getEndpointAddress();

// Instantiate the IoT Agent with your AWS credentials
final String thingName = "RemoteDeviceA";
final String tunnelNotificationTopic = String.format("$aws/things/%s/tunnels/notify",
 thingName);
final AWSIotMqttClient mqttClient = new AWSIotMqttClient(endpoint, thingName,
                 "your_aws_access_key", "your_aws_secret_key");

try {
    mqttClient.connect();
    final TunnelNotificationListener listener = new
 TunnelNotificationListener(tunnelNotificationTopic);
    mqttClient.subscribe(listener, true);
}
finally {
    mqttClient.disconnect();
}

private static class TunnelNotificationListener extends AWSIotTopic {
    public TunnelNotificationListener(String topic) {
        super(topic);
    }

    @Override
    public void onMessage(AWSIotMessage message) {
        try {
            // Deserialize the MQTT message
            final JSONObject json = new JSONObject(message.getStringPayload());
```

```
            final String accessToken = json.getString("clientAccessToken");
            final String region = json.getString("region");

            final String clientMode = json.getString("clientMode");
            if (!clientMode.equals("destination")) {
                throw new RuntimeException("Client mode " + clientMode + " in the MQTT
 message is not expected");
            }

            final JSONArray servicesArray = json.getJSONArray("services");
            if (servicesArray.length() > 1) {
                throw new RuntimeException("Services in the MQTT message has more than
 1 service");
            }
            final String service = servicesArray.get(0).toString();
            if (!service.equals("SSH")) {
                throw new RuntimeException("Service " + service + " is not supported");
            }

            // Start the destination local proxy in a separate process to connect to
 the SSH Daemon listening port 22
            final ProcessBuilder pb = new ProcessBuilder("localproxy",
                        "-t", accessToken,
                        "-r", region,
                        "-d", "localhost:22");
            pb.start();
        }
        catch (Exception e) {
            log.error("Failed to start the local proxy", e);
        }
    }
}
```

Controlling access to tunnels

Secure tunneling provides service-specific actions, resources, and condition context keys for use in IAM permissions policies.

Tunnel access prerequisites

- Learn how to secure AWS resources by using IAM policies.
- Learn how to create and evaluate IAM conditions.

- Learn how to secure AWS resources using resource tags.

Tunnel access policies

You must use the following policies for authorizing permissions to use the secure tunneling API. For more information about AWS IoT security see Identity and access management for AWS IoT.

iot:OpenTunnel

The `iot:OpenTunnel` policy action grants a principal permission to call OpenTunnel.

In the `Resource` element of the IAM policy statement:

- Specify the wildcard tunnel ARN:

  `arn:aws:iot:aws-region:aws-account-id:tunnel/*`
- Specify a thing ARN to manage the `OpenTunnel` permission for specific IoT things:

  `arn:aws:iot:aws-region:aws-account-id:thing/thing-name`

For example, the following policy statement allows you to open a tunnel to the IoT thing named `TestDevice`.

```
{
    "Effect": "Allow",
    "Action": "iot:OpenTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*",
        "arn:aws:iot:aws-region:aws-account-id:thing/TestDevice"
    ]
}
```

The `iot:OpenTunnel` policy action supports the following condition keys:

- `iot:ThingGroupArn`
- `iot:TunnelDestinationService`
- `aws:RequestTag/tag-key`
- `aws:SecureTransport`

- aws:TagKeys

The following policy statement allows you to open a tunnel to the thing if the thing belongs to a thing group with a name that starts with TestGroup and the configured destination service on the tunnel is SSH.

```
{
    "Effect": "Allow",
    "Action": "iot:OpenTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "ForAnyValue:StringLike": {
            "iot:ThingGroupArn": [
                "arn:aws:iot:aws-region:aws-account-id:thinggroup/TestGroup*"
            ]
        },
        "ForAllValues:StringEquals": {
            "iot:TunnelDestinationService": [
                "SSH"
            ]
        }
    }
}
```

You can also use resource tags to control permission to open tunnels. For example, the following policy statement allows a tunnel to be opened if the tag key Owner is present with a value of Admin and no other tags are specified. For general information about using tags, see Tagging your AWS IoT resources.

```
{
    "Effect": "Allow",
    "Action": "iot:OpenTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:RequestTag/Owner": "Admin"
        },
```

```
            "ForAllValues:StringEquals": {
                "aws:TagKeys": "Owner"
            }
        }
}
```

iot:RotateTunnelAccessToken

The `iot:RotateTunnelAccessToken` policy action grants a principal permission to call [RotateTunnelAccessToken](https://docs.aws.amazon.com/iot/latest/apireference/API_iot-secure-tunneling_RotateTunnelAccessToken.html).

In the `Resource` element of the IAM policy statement:

- Specify a fully qualified tunnel ARN:

  `arn:aws:iot:aws-region: aws-account-id:tunnel/tunnel-id`

  You can also use the wildcard tunnel ARN:

  `arn:aws:iot:aws-region:aws-account-id:tunnel/*`
- Specify a thing ARN to manage the `RotateTunnelAccessToken` permission for specific IoT things:

  `arn:aws:iot:aws-region:aws-account-id:thing/thing-name`

For example, the following policy statement allows you to rotate either a tunnel's source access token or a client's destination access token for the IoT thing named `TestDevice`.

```
{
    "Effect": "Allow",
    "Action": "iot:RotateTunnelAccessToken",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*",
        "arn:aws:iot:aws-region:aws-account-id:thing/TestDevice"
    ]
}
```

The `iot:RotateTunnelAccessToken` policy action supports the following condition keys:

- `iot:ThingGroupArn`
- `iot:TunnelDestinationService`

- `iot:ClientMode`
- `aws:SecureTransport`

The following policy statement allows you to rotate the destination access token to the thing if the thing belongs to a thing group with a name that starts with `TestGroup`, the configured destination service on the tunnel is SSH, and the client is in `DESTINATION` mode.

```
{
    "Effect": "Allow",
    "Action": "iot:RotateTunnelAccessToken",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "ForAnyValue:StringLike": {
            "iot:ThingGroupArn": [
                "arn:aws:iot:aws-region:aws-account-id:thinggroup/TestGroup*"
            ]
        },
        "ForAllValues:StringEquals": {
            "iot:TunnelDestinationService": [
                "SSH"
            ],
            "iot:ClientMode": "DESTINATION"
        }
    }
}
```

iot:DescribeTunnel

The `iot:DescribeTunnel` policy action grants a principal permission to call [DescribeTunnel](https://docs.aws.amazon.com/iot/latest/apireference/API_iot-secure-tunneling_DescribeTunnel.html).

In the `Resource` element of the IAM policy statement, specify a fully qualified tunnel ARN:

`arn:aws:iot:`*`aws-region`*`:` *`aws-account-id`*`:tunnel/`*`tunnel-id`*

You can also use the wildcard ARN:

`arn:aws:iot:`*`aws-region`*`:`*`aws-account-id`*`:tunnel/*`

The `iot:DescribeTunnel` policy action supports the following condition keys:

- aws:ResourceTag/*tag-key*
- aws:SecureTransport

The following policy statement allows you to call DescribeTunnel if the requested tunnel is tagged with the key Owner with a value of Admin.

```
{
    "Effect": "Allow",
    "Action": "iot:DescribeTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:ResourceTag/Owner": "Admin"
        }
    }
}
```

iot:ListTunnels

The iot:ListTunnels policy action grants a principal permission to call ListTunnels.

In the Resource element of the IAM policy statement:

- Specify the wildcard tunnel ARN:

  arn:aws:iot:*aws-region*:*aws-account-id*:tunnel/*
- Specify a thing ARN to manage the ListTunnels permission on selected IoT things:

  arn:aws:iot:*aws-region*:*aws-account-id*:thing/*thing-name*

The iot:ListTunnels policy action supports the condition key aws:SecureTransport.

The following policy statement allows you to list tunnels for the thing named TestDevice.

```
{
    "Effect": "Allow",
    "Action": "iot:ListTunnels",
    "Resource": [
```

```
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*",
        "arn:aws:iot:aws-region:aws-account-id:thing/TestDevice"
    ]
}
```

iot:ListTagsForResource

The `iot:ListTagsForResource` policy action grants a principal permission to call `ListTagsForResource`.

In the `Resource` element of the IAM policy statement, specify a fully qualified tunnel ARN:

`arn:aws:iot:aws-region: aws-account-id:tunnel/tunnel-id`

You can also use the wildcard tunnel ARN:

`arn:aws:iot:aws-region:aws-account-id:tunnel/*`

The `iot:ListTagsForResource` policy action supports the condition key `aws:SecureTransport`.

iot:CloseTunnel

The `iot:CloseTunnel` policy action grants a principal permission to call CloseTunnel.

In the `Resource` element of the IAM policy statement, specify a fully qualified tunnel ARN:

`arn:aws:iot:aws-region: aws-account-id:tunnel/tunnel-id`

You can also use the wildcard tunnel ARN:

`arn:aws:iot:aws-region:aws-account-id:tunnel/*`

The `iot:CloseTunnel` policy action supports the following condition keys:

- `iot:Delete`
- `aws:ResourceTag/tag-key`
- `aws:SecureTransport`

The following policy statement allows you to call `CloseTunnel` if the request's `Delete` parameter is `false` and the requested is tagged with the key `Owner` with a value of `QATeam`.

```
{
    "Effect": "Allow",
    "Action": "iot:CloseTunnel",
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:tunnel/*"
    ],
    "Condition": {
        "Bool": {
            "iot:Delete": "false"
        },
        "StringEquals": {
            "aws:ResourceTag/Owner": "QATeam"
        }
    }
}
```

iot:TagResource

The `iot:TagResource` policy action grants a principal permission to call `TagResource`.

In the `Resource` element of the IAM policy statement, specify a fully qualified tunnel ARN:

`arn:aws:iot:aws-region: aws-account-id:tunnel/tunnel-id`

You can also use the wildcard tunnel ARN:

`arn:aws:iot:aws-region:aws-account-id:tunnel/*`

The `iot:TagResource` policy action supports the condition key `aws:SecureTransport`.

iot:UntagResource

The `iot:UntagResource` policy action grants a principal permission to call `UntagResource`.

In the `Resource` element of the IAM policy statement, specify a fully qualified tunnel ARN:

`arn:aws:iot:aws-region: aws-account-id:tunnel/tunnel-id`

You can also use the wildcard tunnel ARN:

`arn:aws:iot:aws-region:aws-account-id:tunnel/*`

The `iot:UntagResource` policy action supports the condition key `aws:SecureTransport`.

Resolving AWS IoT secure tunneling connectivity issues by rotating client access tokens

When you use AWS IoT secure tunneling, you might run into connectivity issues even if the tunnel is open. The following sections show some possible issues and how you can resolve them by rotating the client access tokens. To rotate the client access token (CAT), use the RotateTunnelAccessToken API or the rotate-tunnel-access-token AWS CLI. Depending on whether you run into an error with using the client in the source or destination mode, you can rotate the CAT either in source or destination mode, or both.

Note

- If you're not sure whether the CAT needs to be rotated on the source or destination, you can rotate the CAT on both the source and destination by setting `ClientMode` to ALL when using the `RotateTunnelAccessToken` API.
- Rotating the CAT doesn't extend the tunnel duration. For example, say the tunnel duration is 12 hours and the tunnel has already been open for 4 hours. When you rotate the access tokens, the new tokens that are generated can only be used for the remaining 8 hours.

Topics

Invalid client access token error

When using AWS IoT secure tunneling, you can run into a connection error when using the same client access token (CAT) to reconnect to the same tunnel. In this case, the local proxy can't connect to the secure tunneling proxy server. If you're using a client in the source mode, you might see the following error message:

```
Invalid access token: The access token was previously used and cannot be used again
```

The error occurs because the client access token (CAT) can only be used once by the local proxy, and it then becomes invalid. To resolve this error, rotate the client access token in the SOURCE mode to generate a new CAT for the source. For an example that shows how to rotate the source CAT, see Rotate source CAT example.

Client token mismatch error

> **Note**
>
> Using client tokens to reuse the CAT is not recommended. We recommend that you use the `RotateTunnelAccessToken` API instead to rotate the client access tokens to reconnect to the tunnel.

If you're using client tokens, you can reuse the CAT for reconnecting to the tunnel. To reuse the CAT, you must provide the client token with the CAT the first time you connect to secure tunneling. Secure tunneling stores the client token so for subsequent connection attempts using the same token, the client token must also be provided. For more information about using client tokens, see the local proxy reference implementation in GitHub.

When using client tokens, if you're using a client in the source mode, you might see the following error:

```
Invalid client token: The provided client token does not match the client token
    that was previously set.
```

The error occurs because the client token provided doesn't match the client token that was provided with the CAT when accessing the tunnel. To resolve this error, rotate the CAT in the SOURCE mode to generate a new CAT for the source. The following shows an example:

Rotate source CAT example

The following shows an example of how to run the `RotateTunnelAccessToken` API in the SOURCE mode to generate a new CAT for the source:

```
aws iotsecuretunneling rotate-tunnel-access-token \
    --region <region> \
    --tunnel-id <tunnel-id> \
```

```
    --client-mode SOURCE
```

Running this command generates a new source access token and returns the ARN of your tunnel.

```
{
    "sourceAccessToken": "<source-access-token>",
    "tunnelArn": "arn:aws:iot:<region>:<account-id>:tunnel/<tunnel-id>"
}
```

You can now use the new source token to connect the local proxy in source mode.

```
export AWSIOT_TUNNEL_ACCESS_TOKEN=<source-access-token>
./localproxy -r <region> -s <port>
```

The following shows a sample output of running the local proxy:

```
...

[info]    Starting proxy in source mode
...
[info]    Successfully established websocket connection with proxy server ...
[info]    Listening for new connection on port <port>
...
```

Remote device connectivity issues

When using AWS IoT secure tunneling, the device might get disconnected unexpectedly even if the tunnel is open. To identify whether a device is still connected to the tunnel, you can use the DescribeTunnel API or the describe-tunnel AWS CLI.

A device can get disconnected for multiple reasons. To resolve the connectivity issue, you can rotate the CAT on the destination if the device was disconnected due to the following possible reasons:

- The CAT on the destination became invalid.
- The token wasn't delivered to the device over the secure tunneling reserved MQTT topic:

  ```
  $aws/things/<thing-name>/tunnels/notify
  ```

The following example shows how to resolve this issue:

Rotate destination CAT example

Consider a remote device, *<RemoteThing1>*. To open a tunnel for that thing, you can use the following command:

```
aws iotsecuretunneling open-tunnel \
    --region <region> \
    --destination-config thingName=<RemoteThing1>,services=SSH
```

Running this command generates the tunnel details and the CAT for your source and destination.

```
{
    "sourceAccessToken": "<source-access-token>",
    "destinationAccessToken": "<destination-access-token>",
    "tunnelId": "<tunnel-id>",
    "tunnelArn": "arn:aws:iot:<region>:<account-id>:tunnel/tunnel-id"
}
```

However, when you use the [DescribeTunnel](https://docs.aws.amazon.com/iot/latest/apireference/API_iot-secure-tunneling_DescribeTunnel.html) API, the output indicates that the device has been disconnected, as illustrated below:

```
aws iotsecuretunneling describe-tunnel \
    --tunnel-id <tunnel-id> \
    --region <region>
```

Running this command displays that the device is still not connected.

```
{
    "tunnel": {
        ...
        "destinationConnectionState": {
            "status": "DISCONNECTED"
        },
        ...
    }
}
```

To resolve this error, run the `RotateTunnelAccessToken` API with the client in `DESTINATION` mode and the configurations for the destination. Running this command revokes the old access token, generates a new token, and resends this token to the MQTT topic:

```
$aws/things/<thing-name>/tunnels/notify
```

```
aws iotsecuretunneling rotate-tunnel-access-token \
    --tunnel-id <tunnel-id> \
    --client-mode DESTINATION \
    --destination-config thingName=<RemoteThing1>,services=SSH \
    --region <region>
```

Running this command generates the new access token as shown below. The token is then delivered to the device to connect to the tunnel, if the device agent is set up correctly.

```
{
    "destinationAccessToken": "destination-access-token",
    "tunnelArn": "arn:aws:iot:region:account-id:tunnel/tunnel-id"
}
```

Device provisioning

AWS provides several different ways to provision a device and install unique client certificates on it. This section describes each way and how to select the best one for your IoT solution. These options are described in detail in the whitepaper titled Device Manufacturing and Provisioning with X.509 Certificates in AWS IoT Core.

Select the option that fits your situation best

- **You can install certificates on IoT devices before they are delivered**

  If you can securely install unique client certificates on your IoT devices before they are delivered for use by the end user, you want to use *just-in-time* provisioning (JITP) or *just-in-time* registration (JITR).

  Using JITP and JITR, the certificate authority (CA) used to sign the device certificate is registered with AWS IoT and is recognized by AWS IoT when the device first connects. The device is provisioned in AWS IoT on its first connection using the details of its provisioning template.

  For more information on single thing, JITP, JITR, and bulk provisioning of devices that have unique certificates, see the section called “Provisioning devices that have device certificates”.
- **End users or installers can use an app to install certificates on their IoT devices**

  If you cannot securely install unique client certificates on your IoT device before they are delivered to the end user, but the end user or an installer can use an app to register the devices and install the unique device certificates, you want to use the provisioning by trusted user process.

  Using a trusted user, such as an end user or an installer with a known account, can simplify the device manufacturing process. Instead of a unique client certificate, devices have a temporary certificate that enables the device to connect to AWS IoT for only 5 minutes. During that 5-minute window, the trusted user obtains a unique client certificate with a longer life and installs it on the device. The limited life of the claim certificate minimizes the risk of a compromised certificate.

  For more information, see the section called “Provisioning by trusted user”.
- **End users CANNOT use an app to install certificates on their IoT devices**

If neither of the previous options will work in your IoT solution, the provisioning by claim process is an option. With this process, your IoT devices have a claim certificate that is shared by other devices in the fleet. The first time a device connects with a claim certificate, AWS IoT registers the device using its provisioning template and issues the device its unique client certificate for subsequent access to AWS IoT.

This option enables automatic provisioning of a device when it connects to AWS IoT, but could present a larger risk in the event of a compromised claim certificate. If a claim certificate becomes compromised, you can deactivate the certificate. Deactivating the claim certificate prevents all devices with that claim certificate from being registered in the future. However; deactivating the claim certificate does not block devices that have already been provisioned.

For more information, see the section called “Provisioning by claim”.

Provisioning devices in AWS IoT

When you provision a device with AWS IoT, you must create resources so your devices and AWS IoT can communicate securely. Other resources can be created to help you manage your device fleet. The following resources can be created during the provisioning process:

- An IoT thing.

  IoT things are entries in the AWS IoT device registry. Each thing has a unique name and set of attributes, and is associated with a physical device. Things can be defined using a thing type or grouped into thing groups. For more information, see Managing devices with AWS IoT.

  Although not required, creating a thing makes it possible to manage your device fleet more effectively by searching for devices by thing type, thing group, and thing attributes. For more information, see Fleet indexing.

  > **Note**
  >
  > To index your Thing's connectivity status data, provision your Thing and configure it so the Thing name matches the client ID used on the Connect request.

- An X.509 certificate.

Devices use X.509 certificates to perform mutual authentication with AWS IoT. You can register an existing certificate or have AWS IoT generate and register a new certificate for you. You associate a certificate with a device by attaching it to the thing that represents the device. You must also copy the certificate and associated private key onto the device. Devices present the certificate when connecting to AWS IoT. For more information, see Authentication.

- An IoT policy.

  IoT policies define the operations that a device can perform in AWS IoT. IoT policies are attached to device certificates. When a device presents the certificate to AWS IoT, it is granted the permissions specified in the policy. For more information, see Authorization. Each device needs a certificate to communicate with AWS IoT.

AWS IoT supports automated fleet provisioning using provisioning templates. Provisioning templates describe the resources AWS IoT requires to provision your device. Templates contain variables that enable you to use one template to provision multiple devices. When you provision a device, you specify values for the variables specific to the device using a dictionary or *map*. To provision another device, specify new values in the dictionary.

You can use automated provisioning whether or not your devices have unique certificates (and their associated private key).

Fleet provisioning APIs

There are several categories of APIs used in fleet provisioning:

- These control plane functions create and manage the fleet provisioning templates and configure trusted user policies.
  - CreateProvisioningTemplate
  - CreateProvisioningTemplateVersion
  - DeleteProvisioningTemplate
  - DeleteProvisioningTemplateVersion
  - DescribeProvisioningTemplate
  - DescribeProvisioningTemplateVersion
  - ListProvisioningTemplates
  - ListProvisioningTemplateVersions

  - UpdateProvisioningTemplate
- Trusted users can use this control plane function to generate a temporary onboarding claim. This temporary claim is passed to the device during Wi-Fi configuration or a similar method.
  - CreateProvisioningClaim
- The MQTT API used during the provisioning process by devices with a provisioning claim certificate embedded in a device, or passed to it by a trusted user.
  - the section called “CreateCertificateFromCsr”
  - the section called “CreateKeysAndCertificate”
  - the section called “RegisterThing”

Provisioning devices that don't have device certificates using fleet provisioning

By using AWS IoT fleet provisioning, AWS IoT can generate and securely deliver device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

There are two ways to use fleet provisioning:

- Provisioning by claim
- Provisioning by trusted user

Provisioning by claim

Devices can be manufactured with a provisioning claim certificate and private key (which are special purpose credentials) embedded in them. If these certificates are registered with AWS IoT, the service can exchange them for unique device certificates that the device can use for regular operations. This process includes the following steps:

Before you deliver the device

1. Call `CreateProvisioningTemplate` to create a provisioning template. This API returns a template ARN. For more information, see Device provisioning MQTT API.

   You can also create a fleet provisioning template in the AWS IoT console.

a. From the navigation pane, choose the **Connect many devices** dropdown. Then, choose **Connect many devices**.

b. Choose **Create provisioning template**.

c. Choose the **Provisioning scenario** that best fits your installation processes. Then, choose **Next**.

d. Complete the template workflow.

2. Create certificates and associated private keys to be used as provisioning claim certificates.

3. Register these certificates with AWS IoT and associate an IoT policy that restricts the use of the certificates. The following example IoT policy restricts the use of the certificate associated with this policy to provisioning devices.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/$aws/certificates/create/
*",
    "arn:aws:iot:us-east-1:123456789012:topic/$aws/provisioning-
templates/templateName/provision/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
```

```
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/certificates/
create/*",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/provisioning-
templates/templateName/provision/*"
            ]
        }
    ]
}
```

4. Give the AWS IoT service permission to create or update IoT resources such as things and certificates in your account when provisioning devices. Do this by attaching the `AWSIoTThingsRegistration` managed policy to an IAM role (called the provisioning role) that trusts the AWS IoT service principal.
5. Manufacture the device with the provisioning claim certificate securely embedded in it.

The device is now ready to be delivered to where it will be installed for use.

> **Important**
>
> Provisioning claim private keys should be secured at all times, including on the device. We recommend that you use AWS IoT CloudWatch metrics and logs to monitor for indications of misuse. If you detect misuse, turn off the provisioning claim certificate so it cannot be used for device provisioning.

To initialize the device for use

1. The device uses the AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client to connect to and authenticate with AWS IoT using the provisioning claim certificate that is installed on the device.

   > **Note**
   >
   > For security, the `certificateOwnershipToken` returned by `CreateCertificateFromCsr` and `CreateKeysAndCertificate` expires after one hour. `RegisterThing` must be called before the `certificateOwnershipToken` expires. If the certificate created by `CreateCertificateFromCsr` or `CreateKeysAndCertificate` has not been activated and has not been attached to a policy or a thing by the time the token expires, the certificate is deleted.

If the token expires, the device can call `CreateCertificateFromCsr` or `CreateKeysAndCertificate` again to generate a new certificate.

2. The device obtains a permanent certificate and private key by using one of these options. The device will use the certificate and key for all future authentication with AWS IoT.

   a. Call `CreateKeysAndCertificate` to create a new certificate and private key using the AWS certificate authority.

      Or

   b. Call `CreateCertificateFromCsr` to generate a certificate from a certificate signing request that keeps its private key secure.

3. From the device, call `RegisterThing` to register the device with AWS IoT and create cloud resources.

   The Fleet Provisioning service uses a provisioning template to define and create cloud resources such as IoT things. The template can specify attributes and groups that the thing belongs to. The thing groups must exist before the new thing can be added to them.

4. After saving the permanent certificate on the device, the device must disconnect from the session that it initiated with the provisioning claim certificate and reconnect using the permanent certificate.

The device is now ready to communicate normally with AWS IoT.

Provisioning by trusted user

In many cases, a device connects to AWS IoT for the first time when a trusted user, such as an end user or installation technician, uses a mobile app to configure the device in its deployed location.

> **Important**
>
> You must manage the trusted user's access and permission to perform this procedure. One way to do this is to provide and maintain an account for the trusted user that authenticates them and grants them access to the AWS IoT features and API operations required to perform this procedure.

Before you deliver the device

1. Call `CreateProvisioningTemplate` to create a provisioning template and return its *templateArn* and *templateName*.
2. Create an IAM role that is used by a trusted user to initiate the provisioning process. The provisioning template allows only that user to provision a device. For example:

```
{
    "Effect": "Allow",
    "Action": [
        "iot:CreateProvisioningClaim"
    ],
    "Resource": [
        "arn:aws:iot:aws-region:aws-account-id:provisioningtemplate/templateName"
    ]
}
```

3. Give the AWS IoT service permission to create or update IoT resources, such as things and certificates in your account when provisioning devices. You do this by attaching the `AWSIoTThingsRegistration` managed policy to an IAM role (called the *provisioning role*) that trusts the AWS IoT service principal.
4. Provide the means to identify your trusted users, such as by providing them with an account that can authenticate them and authorize their interactions with the AWS API operations necessary to register their devices.

To initialize the device for use

1. A trusted user signs in to your provisioning mobile app or web service.
2. The mobile app or web application uses the IAM role and calls `CreateProvisioningClaim` to obtain a temporary provisioning claim certificate from AWS IoT.

> **Note**
>
> For security, the temporary provisioning claim certificate that `CreateProvisioningClaim` returns expires after five minutes. The following steps must successfully return a valid certificate before the temporary provisioning claim certificate expires. Temporary provisioning claim certificates do not appear in your account's list of certificates.

3. The mobile app or web application supplies the temporary provisioning claim certificate to the device along with any required configuration information, such as Wi-Fi credentials.
4. The device uses the temporary provisioning claim certificate to connect to AWS IoT using the AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client.
5. The device obtains a permanent certificate and private key by using one of these options within five minutes of connecting to AWS IoT with the temporary provisioning claim certificate. The device will use the certificate and key these options return for all future authentication with AWS IoT.

    a. Call `CreateKeysAndCertificate` to create a new certificate and private key using the AWS certificate authority.

    Or

    b. Call `CreateCertificateFromCsr` to generate a certificate from a certificate signing request that keeps its private key secure.

> **Note**
>
> Remember `CreateKeysAndCertificate` or `CreateCertificateFromCsr` must return a valid certificate within five minutes of connecting to AWS IoT with the temporary provisioning claim certificate.

6. The device calls `RegisterThing` to register the device with AWS IoT and create cloud resources.

    The Fleet Provisioning service uses a provisioning template to define and create cloud resources such as IoT things. The template can specify attributes and groups that the thing belongs to. The thing groups must exist before the new thing can be added to them.

7. After saving the permanent certificate on the device, the device must disconnect from the session that it initiated with the temporary provisioning claim certificate and reconnect using the permanent certificate.

The device is now ready to communicate normally with AWS IoT.

Using pre-provisioning hooks with the AWS CLI

The following procedure creates a provisioning template with pre-provisioning hooks. The Lambda function used here is an example that can be modified.

To create and apply a pre-provisioning hook to a provisioning template

1. Create a Lambda function that has a defined input and output. Lambda functions are highly customizable. `allowProvisioning` and `parameterOverrides` are required for creating pre-provisioning hooks. For more information about creating Lambda functions, see [Using AWS Lambda with the AWS Command Line Interface](#).

   The following is an example of a Lambda function output:

   ```
   {
     "allowProvisioning": True,
     "parameterOverrides": {
       "incomingKey0": "incomingValue0",
       "incomingKey1": "incomingValue1"
     }
   }
   ```

2. AWS IoT uses resource-based policies to call Lambda, so you must give AWS IoT permission to call your Lambda function.

   > **Important**
   >
   > Be sure to include the `source-arn` or `source-account` in the global condition context keys of the policies attached to your Lambda action to prevent permission manipulation. For more information about this, see [Cross-service confused deputy prevention](#).

   The following is an example using [add-permission](#) give IoT permission to your Lambda.

   ```
   aws lambda add-permission \
       --function-name myLambdaFunction \
       --statement-id iot-permission \
       --action lambda:InvokeFunction \
       --principal iot.amazonaws.com
   ```

3. Add a pre-provisioning hook to a template using either the create-provisioning-template or update-provisioning-template command.

   The following CLI example uses the create-provisioning-template to create a provisioning template that has pre-provisioning hooks:

```
aws iot create-provisioning-template \
    --template-name myTemplate \
    --provisioning-role-arn arn:aws:iam:us-east-1:1234564789012:role/myRole \
    --template-body file://template.json \
    --pre-provisioning-hook file://hooks.json
```

   The output of this command looks like the following:

```
{
    "templateArn": "arn:aws:iot:us-east-1:1234564789012:provisioningtemplate/
myTemplate",
    "defaultVersionId": 1,
    "templateName": myTemplate
}
```

   You can also load a parameter from a file instead of typing it all as a command line parameter value to save time. For more information, see Loading AWS CLI Parameters from a File. The following shows the `template` parameter in expanded JSON format:

```
{
    "Parameters" : {
        "DeviceLocation": {
            "Type": "String"
        }
    },
    "Mappings": {
        "LocationTable": {
            "Seattle": {
                "LocationUrl": "https://example.aws"
            }
        }
    },
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
```

```
            "Properties" : {
                "AttributePayload" : {
                    "version" : "v1",
                    "serialNumber" : "serialNumber"
                },
                "ThingName" : {"Fn::Join":["",["ThingPrefix_",
{"Ref":"SerialNumber"}]]},
                "ThingTypeName" : {"Fn::Join":["",["ThingTypePrefix_",
{"Ref":"SerialNumber"}]]},
                "ThingGroups" : ["widgets", "WA"],
                "BillingGroup": "BillingGroup"
            },
            "OverrideSettings" : {
                "AttributePayload" : "MERGE",
                "ThingTypeName" : "REPLACE",
                "ThingGroups" : "DO_NOTHING"
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateId": {"Ref": "AWS::IoT::Certificate::Id"},
                "Status" : "Active",
                "ThingPrincipalType" : "EXCLUSIVE_THING"
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : {
                    "Version": "2012-10-17",
                    "Statement": [{
                        "Effect": "Allow",
                        "Action":["iot:Publish"],
                        "Resource": ["arn:aws:iot:us-east-1:504350838278:topic/foo/
bar"]
                    }]
                }
            }
        }
    },
    "DeviceConfiguration": {
        "FallbackUrl": "https://www.example.com/test-site",
        "LocationUrl": {
```

```
            "Fn::FindInMap": ["LocationTable",{"Ref": "DeviceLocation"},
 "LocationUrl"]}
    }
}
```

The following shows the `pre-provisioning-hook` parameter in expanded JSON format:

```
{
     "targetArn" : "arn:aws:lambda:us-
east-1:765219403047:function:pre_provisioning_test",
     "payloadVersion" : "2020-04-01"
}
```

Provisioning devices that have device certificates

AWS IoT provides three ways to provision devices when they already have a device certificate (and associated private key) on them:

- Single-thing provisioning with a provisioning template. This is a good option if you only need to provision devices one at a time.
- Just-in-time provisioning (JITP) with a template that provisions a device when it first connects to AWS IoT. This is a good option if you need to register large numbers of devices, but you don't have information about them that you can assemble into a bulk provisioning list.
- Bulk registration. This option allows you to specify a list of single-thing provisioning template values that are stored in a file in an S3 bucket. This approach works well if you have a large number of known devices whose desired characteristics you can assemble into a list.

Topics

Single thing provisioning

To provision a thing, use the RegisterThing API or the `register-thing` CLI command. The `register-thing` CLI command takes the following arguments:

--template-body

The provisioning template.

--parameters

A list of name-value pairs for the parameters used in the provisioning template, in JSON format (for example, `{"ThingName" : "MyProvisionedThing", "CSR" : "csr-text"}`).

See Provisioning templates.

RegisterThing or `register-thing` returns the ARNs for the resources and the text of the certificate it created:

```
{
    "certificatePem": "certificate-text",
    "resourceArns": {
    "PolicyLogicalName": "arn:aws:iot:us-
west-2:123456789012:policy/2A6577675B7CD1823E271C7AAD8184F44630FFD7",
    "certificate": "arn:aws:iot:us-west-2:123456789012:cert/
cd82bb924d4c6ccbb14986dcb4f40f30d892cc6b3ce7ad5008ed6542eea2b049",
    "thing": "arn:aws:iot:us-west-2:123456789012:thing/MyProvisionedThing"
    }
}
```

If a parameter is omitted from the dictionary, the default value is used. If no default value is specified, the parameter is not replaced with a value.

Just-in-time provisioning

You can use just-in-time provisioning (JITP) to provision your devices when they first attempt to connect to AWS IoT. To provision the device, you must enable automatic registration and associate a provisioning template with the CA certificate used to sign the device certificate. Provisioning successes and errors are logged as Device provisioning metrics in Amazon CloudWatch.

Topics

JITP overview

When a device attempts to connect to AWS IoT by using a certificate signed by a registered CA certificate, AWS IoT loads the template from the CA certificate and uses it to call RegisterThing. The JITP workflow first registers a certificate with a status value of `PENDING_ACTIVATION`. When the device provisioning flow is complete, the status of the certificate is changed to `ACTIVE`.

AWS IoT defines the following parameters that you can declare and reference in provisioning templates:

- `AWS::IoT::Certificate::Country`
- `AWS::IoT::Certificate::Organization`
- `AWS::IoT::Certificate::OrganizationalUnit`
- `AWS::IoT::Certificate::DistinguishedNameQualifier`
- `AWS::IoT::Certificate::StateName`
- `AWS::IoT::Certificate::CommonName`
- `AWS::IoT::Certificate::SerialNumber`
- `AWS::IoT::Certificate::Id`

The values for these provisioning template parameters are limited to what JITP can extract from the subject field of the certificate of the device being provisioned. The certificate must contain values for all of the parameters in the template body. The `AWS::IoT::Certificate::Id` parameter refers to an internally generated ID, not an ID that is contained in the certificate. You can get the value of this ID using the `principal()` function inside an AWS IoT rule.

> **Note**
>
> You can provision devices using AWS IoT Core just-in-time provisioning (JITP) feature without having to send the entire trust chain on a device's first connection to AWS IoT Core. Presenting the CA certificate is optional, but the device is required to send the Server Name Indication (SNI) extension when it connects to AWS IoT Core.

Example template body

The following JSON file is an example template body of a complete JITP template.

```
{
   "Parameters":{
      "AWS::IoT::Certificate::CommonName":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::SerialNumber":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::Country":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::Id":{
         "Type":"String"
      }
   },
   "Resources":{
      "thing":{
         "Type":"AWS::IoT::Thing",
         "Properties":{
            "ThingName":{
               "Ref":"AWS::IoT::Certificate::CommonName"
            },
            "AttributePayload":{
               "version":"v1",
               "serialNumber":{
                  "Ref":"AWS::IoT::Certificate::SerialNumber"
               }
            },
            "ThingTypeName":"lightBulb-versionA",
            "ThingGroups":[
               "v1-lightbulbs",
               {
                  "Ref":"AWS::IoT::Certificate::Country"
               }
            ]
         },
         "OverrideSettings":{
            "AttributePayload":"MERGE",
            "ThingTypeName":"REPLACE",
            "ThingGroups":"DO_NOTHING"
         }
      },
      "certificate":{
```

```
            "Type":"AWS::IoT::Certificate",
            "Properties":{
               "CertificateId":{
                  "Ref":"AWS::IoT::Certificate::Id"
               },
               "Status":"ACTIVE"
            }
         },
         "policy":{
            "Type":"AWS::IoT::Policy",
            "Properties":{
               "PolicyDocument":"{ \"Version\": \"2012-10-17\",        \"Statement
\": [{ \"Effect\": \"Allow\", \"Action\":[\"iot:Publish\"], \"Resource\":
 [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
            }
         }
      }
}
```

This sample template declares values for the `AWS::IoT::Certificate::CommonName`, `AWS::IoT::Certificate::SerialNumber`, `AWS::IoT::Certificate::Country`, and `AWS::IoT::Certificate::Id` provisioning parameters that are extracted from the certificate and used in the `Resources` section. The JITP workflow then uses this template to perform the following actions:

- Register a certificate and set its status to PENDING_ACTIVE.
- Create one thing resource.
- Create one policy resource.
- Attach the policy to the certificate.
- Attach the certificate to the thing.
- Update the certificate status to ACTIVE.

The device provisioning fails if the certificate doesn't have all of the properties mentioned in the `Parameters` section of the `templateBody`. For example, if `AWS::IoT::Certificate::Country` is included in the template, but the certificate doesn't have a `Country` property, the device provisioning fails.

You can also use CloudTrail to troubleshoot issues with your JITP template. For information about the metrics that are logged in Amazon CloudWatch, see Device provisioning metrics. For more information about provisioning templates, see Provisioning templates.

Note

During the provisioning process, just-in-time provisioning (JITP) calls other AWS IoT control plane API operations. These calls might exceed the AWS IoT Throttling Quotas set for your account and result in throttled calls. Contact AWS Customer Support to raise your throttling quotas if necessary.

Register CA using provisioning template

To register a CA by using a complete provisioning template, follow these steps:

1. Save your provisioning template and the role ARN information like the following example as a JSON file:

```
{
     "templateBody" : "{\r\n    \"Parameters\" : {\r\n
 \"AWS::IoT::Certificate::CommonName\": {\r\n            \"Type\": \"String\"\r
\n        },\r\n        \"AWS::IoT::Certificate::SerialNumber\": {\r\n
  \"Type\": \"String\"\r\n        },\r\n        \"AWS::IoT::Certificate::Country
\": {\r\n            \"Type\": \"String\"\r\n        },\r\n
 \"AWS::IoT::Certificate::Id\": {\r\n            \"Type\": \"String\"\r
\n        }\r\n    },\r\n    \"Resources\": {\r\n        \"thing\": {\r
\n            \"Type\": \"AWS::IoT::Thing\",\r\n            \"Properties
\": {\r\n                \"ThingName\": {\r\n                    \"Ref\":
 \"AWS::IoT::Certificate::CommonName\"\r\n                },\r\n
     \"AttributePayload\": {\r\n                  \"version\": \"v1\",\r\n
                  \"serialNumber\": {\r\n                        \"Ref\":
 \"AWS::IoT::Certificate::SerialNumber\"\r\n                     }\r\n
     },\r\n                \"ThingTypeName\": \"lightBulb-versionA\",\r\n
        \"ThingGroups\": [\r\n                    \"v1-lightbulbs\",\r\n
           {\r\n                        \"Ref\": \"AWS::IoT::Certificate::Country
\"\r\n                    }\r\n                ]\r\n            },\r\n
  \"OverrideSettings\": {\r\n                \"AttributePayload\": \"MERGE\",\r\n
                \"ThingTypeName\": \"REPLACE\",\r\n                \"ThingGroups
\": \"DO_NOTHING\"\r\n            }\r\n        },\r\n        \"certificate\": {\r
\n            \"Type\": \"AWS::IoT::Certificate\",\r\n            \"Properties
\": {\r\n                \"CertificateId\": {\r\n                    \"Ref\":
```

```
 \"AWS::IoT::Certificate::Id\"\r\n                }\r\n                \"Status\":
 \"ACTIVE\"\r\n            },\r\n            \"OverrideSettings\": {\r\n
      \"Status\": \"DO_NOTHING\"\r\n            }\r\n        },\r\n        \"policy
\": {\r\n            \"Type\": \"AWS::IoT::Policy\",\r\n            \"Properties
\": {\r\n                \"PolicyDocument\": \"{ \\\"Version\\\": \\\"2012-10-17
     \\\", \\\"Statement\\\": [{ \\\"Effect\\\": \\\"Allow\\\", \\\"Action\\\":[\\
\"iot:Publish\\\"], \\\"Resource\\\": [\\\"arn:aws:iot:us-east-1:123456789012:topic
\/foo\/bar\\\"] }] }\"\r\n            }\r\n        }\r\n    }\r\n}",
     "roleArn" : "arn:aws:iam::123456789012:role/JITPRole"
}
```

In this example, the value of the `templateBody` field must be a JSON object specified as an escaped string and can use only the values in the preceding list. You can use a variety of tools to create the required JSON output, such as `json.dumps` (Python) or `JSON.stringify` (Node). The value of the `roleARN` field must be the ARN of a role that has the `AWSIoTThingsRegistration` attached to it. Also, your template can use an existing `PolicyName` instead of the inline `PolicyDocument` in the example.

2. Register a CA certificate with the RegisterCACertificate API operation or the `register-ca-certificate` CLI command. You will specify the directory of the provisioning template and role ARN information that you saved in the previous step:

   The following shows an example of how to register a CA certificate in `DEFAULT` mode using the AWS CLI:

   ```
   aws iot register-ca-certificate --ca-certificate file://your-ca-cert --
   verification-cert file://your-verification-cert
                   --set-as-active --allow-auto-registration --registration-config
    file://your-template
   ```

   The following shows an example of how to register a CA certificate in `SNI_ONLY` mode using the AWS CLI:

   ```
   aws iot register-ca-certificate --ca-certificate file://your-ca-cert --certificate-
   mode SNI_ONLY
                    --set-as-active --allow-auto-registration --registration-config
    file://your-template
   ```

   For more information, see Register your CA Certificates.

3. (Optional) Update the settings for a CA certificate by using the UpdateCACertificate API operation or the `update-ca-certificate` CLI command.

   The following shows an example of how to update a CA certificate using the AWS CLI:

```
aws iot update-ca-certificate --certificate-id caCertificateId
                --new-auto-registration-status ENABLE --registration-config
 file://your-template
```

Register CA using provisioning template name

To register a CA by using a provisioning template name, follow these steps:

1. Save your provisioning template body as a JSON file. You can find an example template body in example template body.
2. To create a provisioning template, use the CreateProvisioningTemplate API or the `create-provisioning-template` CLI command:

```
aws iot create-provisioning-template --template-name your-template-name \
        --template-body file://your-template-body.json --type JITP \
        --provisioning-role-arn arn:aws:iam::123456789012:role/test
```

> **Note**
>
> For just-in-time provisioning (JITP), you must specify template type to be `JITP` when creating the provisioning template. For more information about the template type, see CreateProvisioningTemplate in the *AWS API Reference*.

3. To register CA with template name, use the RegisterCACertificate API or the `register-ca-certificate` CLI command:

```
aws iot register-ca-certificate --ca-certificate file://your-ca-cert --
verification-cert file://your-verification-cert \
        --set-as-active --allow-auto-registration --registration-config
 templateName=your-template-name
```

Bulk registration

You can use the `start-thing-registration-task` command to register things in bulk. This command takes a provisioning template, an S3 bucket name, a key name, and a role ARN that allows access to the file in the S3 bucket. The file in the S3 bucket contains the values used to replace the parameters in the template. The file must be a newline-delimited JSON file. Each line contains all of the parameter values for registering a single device. For example:

```
{"ThingName": "foo", "SerialNumber": "123", "CSR": "csr1"}
{"ThingName": "bar", "SerialNumber": "456", "CSR": "csr2"}
```

The following bulk registration-related API operations might be useful:

- ListThingRegistrationTasks: Lists the current bulk thing provisioning tasks.
- DescribeThingRegistrationTask: Provides information about a specific bulk thing registration task.
- StopThingRegistrationTask: Stops a bulk thing registration task.
- ListThingRegistrationTaskReports: Used to check the results and failures for a bulk thing registration task.

> **Note**
>
> - Only one bulk registration operation task can run at a time (per account).
> - Bulk registration operations call other AWS IoT control plane API operations. These calls might exceed the AWS IoT Throttling Quotas in your account and cause throttle errors. Contact AWS Customer Support to raise your AWS IoT throttling quotas, if necessary.

Provisioning templates

A provisioning template is a JSON document that uses parameters to describe the resources your device must use to interact with AWS IoT. A provisioning template contains two sections: `Parameters` and `Resources`. There are two types of provisioning templates in AWS IoT. One is used for just-in-time provisioning (JITP) and bulk registration, and the second is used for fleet provisioning.

Topics

Parameters section

The `Parameters` section declares the parameters used in the `Resources` section. Each parameter declares a name, a type, and an optional default value. The default value is used when the dictionary passed in with the template does not contain a value for the parameter. The `Parameters` section of a template document looks like the following:

```
{
    "Parameters" : {
        "ThingName" : {
            "Type" : "String"
        },
        "SerialNumber" : {
            "Type" : "String"
        },
        "Location" : {
            "Type" : "String",
            "Default" : "WA"
        },
        "CSR" : {
            "Type" : "String"
        }
    }
}
```

This template body snippet declares four parameters: `ThingName`, `SerialNumber`, `Location`, and `CSR`. All of these parameters are of type `String`. The `Location` parameter declares a default value of `"WA"`.

Resources section

The `Resources` section of the template body declares the resources required for your device to communicate with AWS IoT: a thing, a certificate, and one or more IoT policies. Each resource specifies a logical name, a type, and a set of properties.

A logical name allows you to refer to a resource elsewhere in the template.

The type specifies the kind of resource that you are declaring. Valid types are:

- `AWS::IoT::Thing`
- `AWS::IoT::Certificate`
- `AWS::IoT::Policy`

The properties you specify depend on the type of resource you are declaring.

Thing resources

Thing resources are declared using the following properties:

- `ThingName`: String.
- `AttributePayload`: Optional. A list of name-value pairs.
- `ThingTypeName`: Optional. String for an associated thing type for the thing.
- `ThingGroups`: Optional. A list of groups to which the thing belongs.
- `BillingGroup`: Optional. String for an associated billing group name.
- `PackageVersions`: Optional. String for an associated package and version names.

Certificate resources

You can specify certificates in one of the following ways:

- A certificate signing request (CSR).
- A certificate ID of an existing device certificate. (Only certificate IDs can be used with a fleet provisioning template.)
- A device certificate created with a CA certificate registered with AWS IoT. If you have more than one CA certificate registered with the same subject field, you must also pass in the CA certificate used to sign the device certificate.

> **Note**
>
> When you declare a certificate in a template, use only one of these methods. For example, if you use a CSR, you cannot also specify a certificate ID or a device certificate. For more information, see [X.509 client certificates](#).

For more information, see [X.509 Certificate overview](#).

Certificate resources are declared using the following properties:

- `CertificateSigningRequest`: String.
- `CertificateId`: String.
- `CertificatePem`: String.
- `CACertificatePem`: String.
- `Status`: Optional. String that can be `ACTIVE` or `INACTIVE`. Defaults to ACTIVE.
- `ThingPrincipalType`: Optional. String that specifies the type of relationship between the thing and the principal (the certificate).
  - `EXCLUSIVE_THING`: Establishes an exclusive relationship. The principal can only be attached to this specific Thing and no others.
  - `NON_EXCLUSIVE_THING`: Attaches the specified principal to things. You can attach multiple Things to the principal. This is the default value if not specified.

> **Note**
>
> You can also provision devices without device certificates. For more information, see [Provisioning devices that don't have device certificates using fleet provisioning](#).

Examples:

- Certificate specified with a CSR:

```
{
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
```

```
            "CertificateSigningRequest": {"Ref" : "CSR"},
            "Status" : "ACTIVE"
        }
    }
}
```

- Certificate specified with an existing certificate ID:

```
{
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
            "CertificateId": {"Ref" : "CertificateId"}
        }
    }
}
```

- Certificate specified with an existing certificate .pem and CA certificate .pem:

```
{
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
            "CACertificatePem": {"Ref" : "CACertificatePem"},
            "CertificatePem": {"Ref" : "CertificatePem"}
        }
    }
}
```

- Exclusively attach one thing to a principal:

```
{
    "certificate" : {
        "Type" : "AWS::IoT::Certificate",
        "Properties" : {
            "ThingPrincipalType" : "EXCLUSIVE_THING"
        }
    }
}
```

Policy resources

Policy resources are declared using one of the following properties:

- `PolicyName`: Optional. String. Defaults to a hash of the policy document. The `PolicyName` can only reference AWS IoT policies but not IAM policies. If you are using an existing AWS IoT policy, for the `PolicyName` property, enter the name of the policy. Do not include the `PolicyDocument` property.
- `PolicyDocument`: Optional. A JSON object specified as an escaped string. If `PolicyDocument` is not provided, the policy must already be created.

> **Note**
>
> If a `Policy` section is present, `PolicyName` or `PolicyDocument`, but not both, must be specified.

Override settings

If a template specifies a resource that already exists, the `OverrideSettings` section allows you to specify the action to take:

`DO_NOTHING`

Leave the resource as is.

`REPLACE`

Replace the resource with the resource specified in the template.

`FAIL`

Cause the request to fail with a `ResourceConflictsException`.

`MERGE`

Valid only for the `ThingGroups` and `AttributePayload` properties of a `thing`. Merge the existing attributes or group memberships of the thing with those specified in the template.

When you declare a thing resource, you can specify `OverrideSettings` for the following properties:

- ATTRIBUTE_PAYLOAD
- THING_TYPE_NAME
- THING_GROUPS

When you declare a certificate resource, you can specify OverrideSettings for the Status property.

OverrideSettings are not available for policy resources.

Resource example

The following template snippet declares a thing, a certificate, and a policy:

```
{
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "ThingName" : {"Ref" : "ThingName"},
                "AttributePayload" : { "version" : "v1", "serialNumber" :  {"Ref" :
 "SerialNumber"}},
                "ThingTypeName" :  "lightBulb-versionA",
                "ThingGroups" : ["v1-lightbulbs", {"Ref" : "Location"}]
            },
            "OverrideSettings" : {
                "AttributePayload" : "MERGE",
                "ThingTypeName" : "REPLACE",
                "ThingGroups" : "DO_NOTHING"
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateSigningRequest": {"Ref" : "CSR"},
                "Status" : "ACTIVE"
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
```

```
                "PolicyDocument" : "{ \"Version\": \"2012-10-17\",
 \"Statement\": [{ \"Effect\": \"Allow\", \"Action\":[\"iot:Publish\"], \"Resource\":
 [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
            }
        }
    }
}
```

The thing is declared with:

- The logical name "`thing`".
- The type `AWS::IoT::Thing`.
- A set of thing properties.

  The thing properties include the thing name, a set of attributes, an optional thing type name, and an optional list of thing groups to which the thing belongs.

Parameters are referenced by `{"Ref":"`*`parameter-name`*`"}`. When the template is evaluated, the parameters are replaced with the parameter's value from the dictionary passed in with the template.

The certificate is declared with:

- The logical name "`certificate`".
- The type `AWS::IoT::Certificate`.
- A set of properties.

  The properties include the CSR for the certificate, and setting the status to `ACTIVE`. The CSR text is passed as a parameter in the dictionary passed with the template.

The policy is declared with:

- The logical name "`policy`".
- The type `AWS::IoT::Policy`.
- Either the name of an existing policy or a policy document.

Template example for bulk registration

The following JSON file is an example of a complete provisioning template that specifies the certificate with a CSR:

(The `PolicyDocument` field value must be a JSON object specified as an escaped string.)

```
{
    "Parameters" : {
        "ThingName" : {
            "Type" : "String"
        },
        "SerialNumber" : {
            "Type" : "String"
        },
        "Location" : {
            "Type" : "String",
            "Default" : "WA"
        },
        "CSR" : {
            "Type" : "String"
        }
    },
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "ThingName" : {"Ref" : "ThingName"},
                "AttributePayload" : { "version" : "v1", "serialNumber" :  {"Ref" :
 "SerialNumber"}},
                "ThingTypeName" :  "lightBulb-versionA",
                "ThingGroups" : ["v1-lightbulbs", {"Ref" : "Location"}]
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateSigningRequest": {"Ref" : "CSR"},
                "Status" : "ACTIVE",
                "ThingPrincipalType" : "EXCLUSIVE_THING"
            }
        },
        "policy" : {
```

```
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : "{ \"Version\": \"2012-10-17\",
 \"Statement\": [{ \"Effect\": \"Allow\", \"Action\":[\"iot:Publish\"], \"Resource\":
 [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
            }
        }
    }
}
```

Template example for just-in-time provisioning (JITP)

The following JSON file is an example of a complete provisioning template that specifies an existing certificate with a certificate ID:

```
{
   "Parameters":{
      "AWS::IoT::Certificate::CommonName":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::SerialNumber":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::Country":{
         "Type":"String"
      },
      "AWS::IoT::Certificate::Id":{
         "Type":"String"
      }
   },
   "Resources":{
      "thing":{
         "Type":"AWS::IoT::Thing",
         "Properties":{
            "ThingName":{
               "Ref":"AWS::IoT::Certificate::CommonName"
            },
            "AttributePayload":{
               "version":"v1",
               "serialNumber":{
                  "Ref":"AWS::IoT::Certificate::SerialNumber"
               }
            },
```

```
            "ThingTypeName":"lightBulb-versionA",
            "ThingGroups":[
               "v1-lightbulbs",
               {
                  "Ref":"AWS::IoT::Certificate::Country"
               }
            ]
         },
         "OverrideSettings":{
            "AttributePayload":"MERGE",
            "ThingTypeName":"REPLACE",
            "ThingGroups":"DO_NOTHING"
         }
      },
      "certificate":{
         "Type":"AWS::IoT::Certificate",
         "Properties":{
            "CertificateId":{
               "Ref":"AWS::IoT::Certificate::Id"
            },
            "Status":"ACTIVE",
            "ThingPrincipalType" : "EXCLUSIVE_THING"
         }
      },
      "policy":{
         "Type":"AWS::IoT::Policy",
         "Properties":{
            "PolicyDocument":"{ \"Version\": \"2012-10-17\",        \"Statement
\": [{ \"Effect\": \"Allow\", \"Action\":[\"iot:Publish\"], \"Resource\":
 [\"arn:aws:iot:us-east-1:123456789012:topic/foo/bar\"] }] }"
         }
      }
   }
}
```

> **Important**
>
> You must use `CertificateId` in a template that's used for JIT provisioning.

For more information about the type of a provisioning template, see CreateProvisioningTemplate in the AWS API reference.

For more information about how to use this template for just-in-time provisioning, see: Just-in-time provisioning.

Fleet provisioning

Fleet provisioning templates are used by AWS IoT to set up cloud and device configuration. These templates use the same parameters and resources as the JITP and bulk registration templates. For more information, see Provisioning templates. Fleet provisioning templates can contain a `Mapping` section and a `DeviceConfiguration` section. You can use intrinsic functions inside a fleet provisioning template to generate a device-specific configuration. Fleet provisioning templates are named resources and are identified by ARNs (for example, `arn:aws:iot:us-west-2:1234568788:provisioningtemplate/templateName`).

Mappings

The optional `Mappings` section matches a key to a corresponding set of named values. For example, if you want to set values based on an AWS Region, you can create a mapping that uses the AWS Region name as a key and contains the values you want to specify for each specific Region. You use the `Fn::FindInMap` intrinsic function to retrieve values in a map.

You cannot include parameters, pseudo parameters, or call intrinsic functions in the `Mappings` section.

Device configuration

The device configuration section contains arbitrary data that you want to send to your devices when provisioning. For example:

```
{
    "DeviceConfiguration": {
        "Foo":"Bar"
    }
}
```

If you're sending messages to your devices by using the JavaScript Object Notation (JSON) payload format, AWS IoT Core formats this data as JSON. If you're using the Concise Binary Object Representation (CBOR) payload format, AWS IoT Core formats this data as CBOR. The `DeviceConfiguration` section doesn't support nested JSON objects.

Intrinsic functions

Intrinsic functions are used in any section of the provisioning template except the `Mappings` section.

`Fn::Join`

Appends a set of values into a single value, separated by the specified delimiter. If a delimiter is an empty string, the values are concatenated with no delimiter.

> **Important**
>
> `Fn::Join` is not supported for the section called “Policy resources”.

`Fn::Select`

Returns a single object from a list of objects by index.

> **Important**
>
> `Fn::Select` does not check for `null` values or if the index is out of bounds of the array. Both conditions result in a provisioning error, so make sure you chose a valid index value and the list contains non-null values.

`Fn::FindInMap`

Returns the value corresponding to keys in a two-level map that is declared in the `Mappings` section.

`Fn::Split`

Splits a string into a list of string values so you can select an element from the list of strings. You specify a delimiter that determines where the string is split (for example, a comma). After you split a string, use `Fn::Select` to select an element.

For example, if a comma-delimited string of subnet IDs is imported to your stack template, you can split the string at each comma. From the list of subnet IDs, use `Fn::Select` to specify a subnet ID for a resource.

`Fn::Sub`

Substitutes variables in an input string with values that you specify. You can use this function to construct commands or outputs that include values that aren't available until you create or update a stack.

Template example for fleet provisioning

```
{
    "Parameters" : {
        "ThingName" : {
            "Type" : "String"
        },
        "SerialNumber": {
            "Type": "String"
        },
        "DeviceLocation": {
            "Type": "String"
        }
    },
    "Mappings": {
        "LocationTable": {
            "Seattle": {
                "LocationUrl": "https://example.aws"
            }
        }
    },
    "Resources" : {
        "thing" : {
            "Type" : "AWS::IoT::Thing",
            "Properties" : {
                "AttributePayload" : {
                    "version" : "v1",
                    "serialNumber" : "serialNumber"
                },
                "ThingName" : {"Ref" : "ThingName"},
                "ThingTypeName" : {"Fn::Join":["",["ThingPrefix_",
{"Ref":"SerialNumber"}]]},
                "ThingGroups" : ["v1-lightbulbs", "WA"],
                "BillingGroup": "LightBulbBillingGroup"
            },
            "OverrideSettings" : {
```

```
                "AttributePayload" : "MERGE",
                "ThingTypeName" : "REPLACE",
                "ThingGroups" : "DO_NOTHING"
            }
        },
        "certificate" : {
            "Type" : "AWS::IoT::Certificate",
            "Properties" : {
                "CertificateId": {"Ref": "AWS::IoT::Certificate::Id"},
                "Status" : "Active",
                "ThingPrincipalType" : "EXCLUSIVE_THING"
            }
        },
        "policy" : {
            "Type" : "AWS::IoT::Policy",
            "Properties" : {
                "PolicyDocument" : {
                    "Version": "2012-10-17",
                    "Statement": [{
                        "Effect": "Allow",
                        "Action":["iot:Publish"],
                        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/foo/
bar"]
                    }]
                }
            }
        }
    },
    "DeviceConfiguration": {
        "FallbackUrl": "https://www.example.com/test-site",
        "LocationUrl": {
            "Fn::FindInMap": ["LocationTable",{"Ref": "DeviceLocation"},
 "LocationUrl"]}
        }
}
```

Note

An existing provisioning template can be updated to add a pre-provisioning hook.

Pre-provisioning hooks

AWS recommends using pre-provisioning hook functions when creating provisioning templates to allow more control of which and how many devices your account onboards. Pre-provisioning hooks are Lambda functions that validate parameters passed from the device before allowing the device to be provisioned. This Lambda function must exist in your account before you provision a device because it's called every time a device sends a request through the section called “RegisterThing”.

> **Important**
>
> Be sure to include the `source-arn` or `source-account` in the global condition context keys of the policies attached to your Lambda action to prevent permission manipulation. For more information about this, see Cross-service confused deputy prevention.

For devices to be provisioned, your Lambda function must accept the input object and return the output object described in this section. The provisioning proceeds only if the Lambda function returns an object with `"allowProvisioning": True`.

Pre-provision hook input

AWS IoT sends this object to the Lambda function when a device registers with AWS IoT.

```
{
    "claimCertificateId" : "string",
    "certificateId" : "string",
    "certificatePem" : "string",
    "templateArn" : "arn:aws:iot:us-east-1:1234567890:provisioningtemplate/MyTemplate",
    "clientId" : "221a6d10-9c7f-42f1-9153-e52e6fc869c1",
    "parameters" : {
        "string" : "string",
        ...
    }
}
```

The `parameters` object passed to the Lambda function contains the properties in the `parameters` argument passed in the the section called “RegisterThing” request payload.

Pre-provision hook return value

The Lambda function must return a response that indicates whether it has authorized the provisioning request and the values of any properties to override.

The following is an example of a successful response from the pre-provisioning function.

```
{
    "allowProvisioning": true,
    "parameterOverrides" : {
        "Key": "newCustomValue",
        ...
    }
}
```

`"parameterOverrides"` values will be added to `"parameters"` parameter of the the section called "RegisterThing" request payload.

Note

- If the Lambda function fails, the provisioning request fails with `ACCESS_DENIED` and an error is logged to CloudWatch Logs.
- If the Lambda function doesn't return `"allowProvisioning": "true"` in the response, the provisioning request fails with `ACCESS_DENIED`.
- The Lambda function must finish running and return within 5 seconds, otherwise the provisioning request fails.

Pre-provisioning hook Lambda example

Python

An example of a pre-provisioning hook Lambda in Python.

```
import json

def pre_provisioning_hook(event, context):
    print(event)
```

```
    return {
        'allowProvisioning': True,
        'parameterOverrides': {
            'DeviceLocation': 'Seattle'
        }
    }
```

Java

An example of a pre-provisioning hook Lambda in Java.

Handler class:

```
package example;

import java.util.Map;
import java.util.HashMap;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class PreProvisioningHook implements
 RequestHandler<PreProvisioningHookRequest, PreProvisioningHookResponse> {

    public PreProvisioningHookResponse handleRequest(PreProvisioningHookRequest
 object, Context context) {
        Map<String, String> parameterOverrides = new HashMap<String, String>();
        parameterOverrides.put("DeviceLocation", "Seattle");

        PreProvisioningHookResponse response = PreProvisioningHookResponse.builder()
                .allowProvisioning(true)
                .parameterOverrides(parameterOverrides)
                .build();

        return response;
    }

}
```

Request class:

```
package example;

import java.util.Map;
```

```
import lombok.Builder;
import lombok.Data;
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class PreProvisioningHookRequest {
    private String claimCertificateId;
    private String certificateId;
    private String certificatePem;
    private String templateArn;
    private String clientId;
    private Map<String, String> parameters;
}
```

Response class:

```
package example;

import java.util.Map;
import lombok.Builder;
import lombok.Data;
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;


@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class PreProvisioningHookResponse {
    private boolean allowProvisioning;
    private Map<String, String> parameterOverrides;
}
```

JavaScript

An example of a pre-provisioning hook Lambda in JavaScript.

```
exports.handler = function(event, context, callback) {
```

```
    console.log(JSON.stringify(event, null, 2));
    var reply = {
        allowProvisioning: true,
        parameterOverrides: {
            DeviceLocation: 'Seattle'
        }
     };
     callback(null, reply);
}
```

Self-managed certificate signing using AWS IoT Core certificate provider

You can create an AWS IoT Core certificate provider to sign certificate signing requests (CSRs) in AWS IoT fleet provisioning. A certificate provider references a Lambda function and the `CreateCertificateFromCsr` MQTT API for fleet provisioning. The Lambda function accepts a CSR and returns a signed client certificate.

When you don't have a certificate provider with your AWS account, the CreateCertificateFromCsr MQTT API is called in fleet provisioning to generate the certificate from a CSR. After you create a certificate provider, the behavior of the CreateCertificateFromCsr MQTT API will change and all calls to this MQTT API will invoke the certificate provider to issue the certificate.

With AWS IoT Core certificate provider, you can implement solutions that utilize private certificate authorities (CAs) such as AWS Private CA, other publicly trusted CAs, or your own Public Key Infrastructure (PKI) to sign the CSR. In addition, you can use certificate provider to customize your client certificate's fields such as validity periods, signing algorithms, issuers, and extensions.

> **Important**
>
> You can only create one certificate provider per AWS account. The signing behavior change applies to the entire fleet that calls the CreateCertificateFromCsr MQTT API until you delete the certificate provider from your AWS account.

In this topic:

- How self-managed certificate signing works in fleet provisioning
- Certificate provider Lambda function input

- Certificate provider Lambda function return value
- Example Lambda function
- Self-managed certificate signing for fleet provisioning
- AWS CLI commands for certificate provider

How self-managed certificate signing works in fleet provisioning

Key concepts

The following concepts provide details that can help you understand how self-managed certificate signing works in AWS IoT fleet provisioning. For more information, see Provisioning devices that don't have device certificates using fleet provisioning.

AWS IoT fleet provisioning

With AWS IoT fleet provisioning (short for fleet provisioning), AWS IoT Core generates and securely delivers device certificates to your devices when they connect to AWS IoT Core for the first time. You can use fleet provisioning to connect devices that don't have device certificates to AWS IoT Core.

Certificate signing request (CSR)

In the process of fleet provisioning, a device makes a request to AWS IoT Core through the fleet provisioning MQTT APIs. This request includes a certificate signing request (CSR), which will be signed to create a client certificate.

AWS managed certificate signing in fleet provisioning

AWS managed is the default setting for certificate signing in fleet provisioning. With AWS managed certificate signing, AWS IoT Core will sign CSRs using its own CAs.

Self-managed certificate signing in fleet provisioning

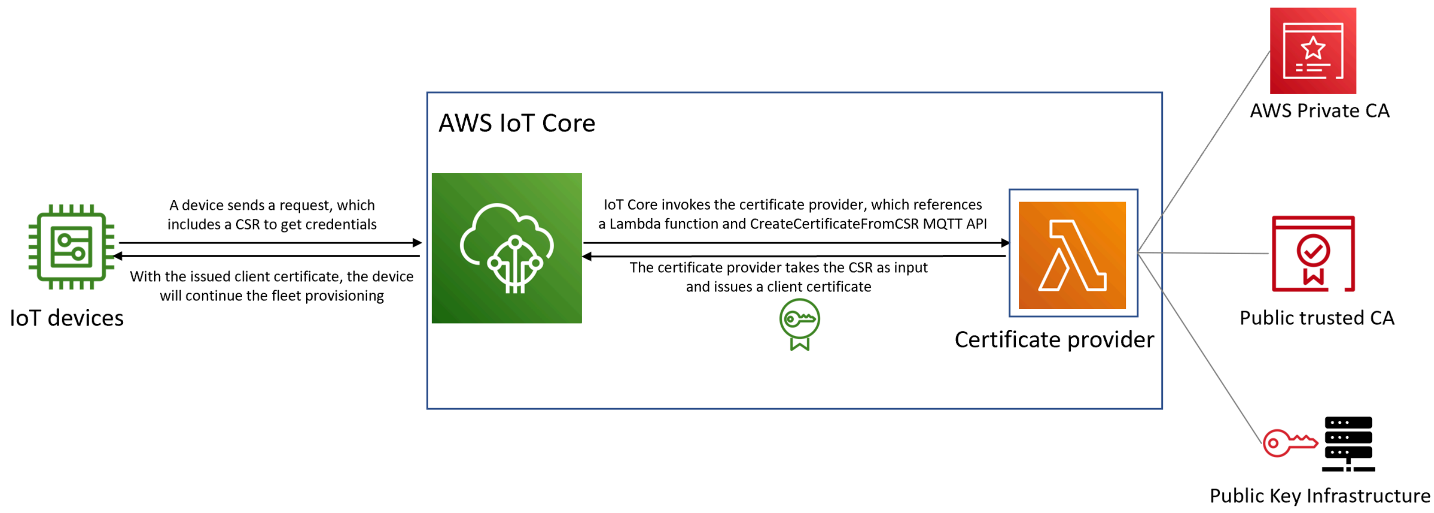
Self-managed is another option for certificate signing in fleet provisioning. With self-managed certificate signing, you create an AWS IoT Core certificate provider to sign CSRs. You can use self-managed certificate signing to sign CSRs with a CA generated by AWS Private CA, other publicly trusted CA, or your own Public Key Infrastructure (PKI).

AWS IoT Core certificate provider

AWS IoT Core certificate provider (short for certificate provider) is a customer-managed resource that's used for self-managed certificate signing in fleet provisioning.

Diagram

The following diagram is a simplified illustration of how self-certificate signing works in AWS IoT fleet provisioning.



- When a new IoT device is manufactured or introduced to the fleet, it needs client certificates to authenticate itself with AWS IoT Core.
- As part of the fleet provisioning process, the device makes a request to AWS IoT Core for client certificates through the fleet provisioning MQTT APIs. This request includes a certificate signing request (CSR).
- AWS IoT Core invokes the certificate provider and passes the CSR as input to the provider.
- The certificate provider takes the CSR as input and issues a client certificate.

  For AWS managed certificate signing, AWS IoT Core signs the CSR using its own CA and issues a client certificate.
- With the issued client certificate, the device will continue the fleet provisioning and establish a secure connection with AWS IoT Core.

Certificate provider Lambda function input

AWS IoT Core sends the following object to the Lambda function when a device registers with it. The value of `certificateSigningRequest` is the CSR in Privacy-Enhanced Mail (PEM) format that's provided in the `CreateCertificateFromCsr` request. The `principalId` is the ID of the principal used to connect to AWS IoT Core when making the `CreateCertificateFromCsr` request. `clientId` is the client ID set for the MQTT connection.

```
{
 "certificateSigningRequest": "string",
 "principalId": "string",
 "clientId": "string"
}
```

Certificate provider Lambda function return value

The Lambda function must return a response that contains the `certificatePem` value. The following is an example of a successful response. AWS IoT Core will use the return value (`certificatePem`) to create the certificate.

```
{
 "certificatePem": "string"
}
```

If the registration is successful, `CreateCertificateFromCsr` will return the same `certificatePem` in the `CreateCertificateFromCsr` response. For more information, see the response payload example of CreateCertificateFromCsr.

Example Lambda function

Before creating a certificate provider, you must create a Lambda function to sign a CSR. The following is an example Lambda function in Python. This function calls AWS Private CA to sign the input CSR, using a private CA and the `SHA256WITHRSA` signing algorithm. The returned client certificate will be valid for one year. For more information about AWS Private CA and how to create a private CA, see What is AWS Private CA? and Creating a private CA.

```
import os
import time
import uuid
import boto3

def lambda_handler(event, context):
    ca_arn = os.environ['CA_ARN']
    csr = (event['certificateSigningRequest']).encode('utf-8')

    acmpca = boto3.client('acm-pca')
    cert_arn = acmpca.issue_certificate(
```

```
        CertificateAuthorityArn=ca_arn,
        Csr=csr,
        Validity={"Type": "DAYS", "Value": 365},
        SigningAlgorithm='SHA256WITHRSA',
        IdempotencyToken=str(uuid.uuid4())
    )['CertificateArn']

    # Wait for certificate to be issued
    time.sleep(1)
    cert_pem = acmpca.get_certificate(
        CertificateAuthorityArn=ca_arn,
        CertificateArn=cert_arn
    )['Certificate']

    return {
        'certificatePem': cert_pem
    }
```

Important

- Certificates returned by the Lambda function must have the same subject name and public key as the Certificate Signing Request (CSR).
- The Lambda function must finish running in 5 seconds.
- The Lambda function must be in the same AWS account and Region as the certificate provider resource.
- The AWS IoT service principal must be granted the invoke permission to the Lambda function. To avoid confused deputy issues, we recommend that you set `sourceArn` and `sourceAccount` for the invoke permissions. For more information, see Cross-service confused deputy prevention.

The following resource-based policy example for Lambda grants AWS IoT the permission to invoke the Lambda function:

```
{
 "Version":"2012-10-17",
 "Id": "InvokePermission",
 "Statement": [
  {
```

```
    "Sid": "LambdaAllowIotProvider",
    "Effect": "Allow",
    "Principal": {
     "Service": "iot.amazonaws.com"
    },
    "Action": "lambda:InvokeFunction",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
    "Condition": {
     "StringEquals": {
      "AWS:SourceAccount": "123456789012"
     },
     "ArnLike": {
     "AWS:SourceArn": "arn:aws:iot:us-east-1:123456789012:certificateprovider/my-
certificate-provider"
     }
    }
   }
  ]
}
```

Self-managed certificate signing for fleet provisioning

You can choose self-managed certificate signing for fleet provisioning using AWS CLI or AWS Management Console.

AWS CLI

To choose self-managed certificate signing, you must create an AWS IoT Core certificate provider to sign CSRs in fleet provisioning. AWS IoT Core invokes the certificate provider, which takes a CSR as input and returns a client certificate. To create a certificate provider, use the `CreateCertificateProvider` API operation or the `create-certificate-provider` CLI command.

> **Note**
>
> After you create a certificate provider, the behavior of [`CreateCertificateFromCsr` API for fleet provisioning] will change so that all calls to `CreateCertificateFromCsr` will invoke the certificate provider to create the certificates. It can take a few minutes for this behavior to change after a certificate provider is created.

```
aws iot create-certificate-provider \
                --certificateProviderName my-certificate-provider \
                --lambdaFunctionArn arn:aws:lambda:us-east-1:123456789012:function:my-function-1 \
                --accountDefaultForOperations CreateCertificateFromCsr
```

The following shows an example output for this command:

```
{
 "certificateProviderName": "my-certificate-provider",
 "certificateProviderArn": "arn:aws:iot:us-east-1:123456789012:certificateprovider:my-certificate-provider"
}
```

For more information, see `CreateCertificateProvider` from the *AWS IoT API Reference*.

AWS Management Console

To choose self-managed certificate signing using AWS Management Console, follow the steps:

1. Go to the AWS IoT console.
2. On the left navigation, under **Security**, choose **Certificate signing**.
3. On the **Certificate signing** page, under **Certificate signing details**, choose **Edit certificate signing method**.
4. On the **Edit certificate signing method** page, under **Certificate signing method**, choose **Self-managed**.
5. In the **Self-managed settings** section, enter a name for certificate provider, then create or choose a Lambda function.
6. Choose **Update certificate signing**.

AWS CLI commands for certificate provider

Create certificate provider

To create a certificate provider, use the `CreateCertificateProvider` API operation or the `create-certificate-provider` CLI command.

Note

After you create a certificate provider, the behavior of [CreateCertificateFromCsr API for fleet provisioning] will change so that all calls to `CreateCertificateFromCsr` will invoke the certificate provider to create the certificates. It can take a few minutes for this behavior to change after a certificate provider is created.

```
aws iot create-certificate-provider \
                --certificateProviderName my-certificate-provider \
                --lambdaFunctionArn arn:aws:lambda:us-east-1:123456789012:function:my-
function-1 \
                --accountDefaultForOperations CreateCertificateFromCsr
```

The following shows an example output for this command:

```
{
 "certificateProviderName": "my-certificate-provider",
 "certificateProviderArn": "arn:aws:iot:us-east-1:123456789012:certificateprovider:my-
certificate-provider"
}
```

For more information, see `CreateCertificateProvider` from the *AWS IoT API Reference*.

Update certificate provider

To update a certificate provider, use the `UpdateCertificateProvider` API operation or the `update-certificate-provider` CLI command.

```
aws iot update-certificate-provider \
                --certificateProviderName my-certificate-provider \
                --lambdaFunctionArn arn:aws:lambda:us-east-1:123456789012:function:my-
function-2 \
                --accountDefaultForOperations CreateCertificateFromCsr
```

The following shows an example output for this command:

```
{
 "certificateProviderName": "my-certificate-provider",
```

```
 "certificateProviderArn": "arn:aws:iot:us-east-1:123456789012:certificateprovider:my-certificate-provider"
}
```

For more information, see UpdateCertificateProvider from the *AWS IoT API Reference*.

Describe certificate provider

To describe a certificate provider, use the `DescribeCertificateProvider` API operation or the `describe-certificate-provider` CLI command.

```
aws iot describe-certificate-provider --certificateProviderName my-certificate-provider
```

The following shows an example output for this command:

```
{
 "certificateProviderName": "my-certificate-provider",
 "lambdaFunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
 "accountDefaultForOperations": [
  "CreateCertificateFromCsr"
 ],
 "creationDate": "2022-11-03T00:15",
 "lastModifiedDate": "2022-11-18T00:15"
}
```

For more information, see DescribeCertificateProvider from the *AWS IoT API Reference*.

Delete certificate provider

To delete a certificate provider, use the `DeleteCertificateProvider` API operation or the `delete-certificate-provider` CLI command. If you delete the certificate provider resource, the behavior of `CreateCertificateFromCsr` will resume, and AWS IoT will create certificates signed by AWS IoT from a CSR.

```
aws iot delete-certificate-provider --certificateProviderName my-certificate-provider
```

This command doesn't produce any output.

For more information, see DeleteCertificateProvider from the *AWS IoT API Reference*.

List certificate provider

To list the certificate providers within your AWS account, use the `ListCertificateProviders` API operation or the `list-certificate-providers` CLI command.

```
aws iot list-certificate-providers
```

The following shows an example output for this command:

```
{
 "certificateProviders": [
  {
   "certificateProviderName": "my-certificate-provider",
   "certificateProviderArn": "arn:aws:iot:us-
east-1:123456789012:certificateprovider:my-certificate-provider"
  }
 ]
}
```

For more information, see `ListCertificateProvider` from the *AWS IoT API Reference*.

Creating IAM policies and roles for a user installing a device

Note

These procedures are for use only when directed by the AWS IoT console.

To go to this page from the console, open create a new provisioning template.

Why can't this be done in the AWS IoT console?

For the most secure experience, IAM actions are performed in the IAM console. The procedures in this section walk you through the steps to create the IAM roles and policies that are needed to use the provisioning template.

Creating an IAM policy for the user who will install a device

This procedure describes how to create an IAM policy that authorizes a user to install a device using a provisioning template.

While performing this procedure, you'll be switching between the IAM console and the AWS IoT console. We recommend having both consoles open at the same time while you complete this procedure.

To create an IAM policy for the user who will install a device

1. Open the Policies hub in the IAM console.
2. Choose **Create Policy**.
3. On the **Create policy** page, choose the **JSON** tab.
4. Switch to the page in the AWS IoT console where you chose **Configure user policy and role**.
5. In the **Sample provisioning policy**, choose **Copy**.
6. Switch back to the IAM console.
7. In the **JSON** editor, paste the policy you copied from the AWS IoT console. This policy is specific to the template you're creating in the AWS IoT console.
8. To continue, choose **Next: Tags**.
9. On the **Add tags (Optional)** page, choose **Add tag** for each tag you want to add to this policy. You can skip this step if you don't have any tags to add.
10. To continue, choose **Next: Review**.
11. On the **Review policy** page, do the following:
    a. For **Name***, enter a name for the policy that will help you remember the policy's purpose.

       Note the name you give this policy because you'll use it in the next procedure.
    b. You can choose to enter an optional description for the policy you're creating.
    c. Review the rest of this policy and its tags.
12. To finish creating the new policy, choose **Create policy**.

After you create your new policy, continue to the section called “Creating an IAM role for the user who will install a device” to create the user's role entry that you'll attach this policy.

Creating an IAM role for the user who will install a device

These steps describe how to create an IAM role that authenticates the user who will install a device using a provisioning template.

To create an IAM policy for the user who will install a device

1. Open the Role hub in the IAM console.
2. Choose **Create role**.
3. In **Select trusted entity**, choose the type of trusted entity that you want to give access to the template you're creating.
4. Choose or enter the identification of the trusted entity that you want to grant access to, and then choose **Next**.
5. On the **Add permissions** page, in **Permission policies**, in the search box, enter the name of the policy you created in the previous procedure.
6. For the policy list, choose the policy that you created in the previous procedure, and then choose **Next**.
7. In the **Name, review, and create** section, do the following:

    a. For **Role name**, enter a role name that will help you remember the role's purpose.

    b. For **Description**, you can choose to enter an optional description of the Role. This isn't required to continue.

    c. Review the values in **Step 1** and **Step 2**.

    d. For **Add tags (Optional)**, you can choose to add tags to this role. This isn't required to continue.

    e. Verify the information on this page is complete and correct, and then choose **Create role**.

After you create the new role, return to the AWS IoT console to continue creating the template.

Updating an existing policy to authorize a new template

The following steps describe how to add a new template to an IAM policy that authorizes a user to install a device using a provisioning template.

To add a new template to an existing IAM policy

1. Open the Policies hub in the IAM console.
2. In the search box, enter the name of the policy to update.
3. For the list below the search box, find the policy you want to update and choose the policy name.

4. For **Policy summary**, choose the **JSON** tab, if that panel isn't already visible.
5. To modify the policy document, choose **Edit policy**.
6. In the editor, choose the **JSON** tab, if that panel isn't already visible.
7. In the policy document, find the policy statement that contains the `iot:CreateProvisioningClaim` action.

   If the policy document doesn't contain a policy statement with the `iot:CreateProvisioningClaim` action, copy the following statement snippet and paste it as an additional entry in the `Statement` array in the policy document.

   > **Note**
   >
   > This snippet must be placed before the closing `]` character in the `Statement` array. You might need to add a comma before or after this snippet to correct any syntax errors.

   ```
   {
       "Effect": "Allow",
       "Action": [
           "iot:CreateProvisioningClaim"
       ],
       "Resource": [
           "--PUT YOUR NEW TEMPLATE ARN HERE--"
       ]
   }
   ```
8. Switch to the page in the AWS IoT console where you chose **Modify user role permissions**.
9. Find the **Resource ARN** of the template and choose **Copy**.
10. Switch back to the IAM console.
11. Paste the copied Amazon Resource Name (ARN) at the top of the list of template ARNs in the `Statement` array so that it's the first entry.

    If this is the only ARN in the array, remove the comma at end of the value you just pasted.
12. Review the updated policy statement and correct any errors indicated by the editor.
13. To save the updated policy document, choose **Review policy**.
14. Review the policy and then choose **Save changes**.

15. Return to the AWS IoT console.

Device provisioning MQTT API

The Fleet Provisioning service supports the following MQTT API operations:

- the section called "CreateCertificateFromCsr"
- the section called "CreateKeysAndCertificate"
- the section called "RegisterThing"

This API supports response buffers in Concise Binary Object Representation (CBOR) format and JavaScript Object Notation (JSON), depending on the *payload-format* of the topic. For clarity, the response and request examples in this section are shown in JSON format.

payload-format	Response format data type
cbor	Concise Binary Object Representation (CBOR)
json	JavaScript Object Notation (JSON)

Important

Before publishing a request message topic, subscribe to the response topics to receive the response. The messages used by this API use MQTT's publish/subscribe protocol to provide a request and response interaction.

If you don't subscribe to the response topics *before* you publish a request, you might not receive the results of that request.

IoT Core Fleet Provisioning returns the device provisioning MQTT API results through the same MQTT connection used to publish the API request.

CreateCertificateFromCsr

Creates a certificate from a certificate signing request (CSR). AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a

PENDING_ACTIVATION status. When you call RegisterThing to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.

For more information on creating a client certificate using your Certificate Authority certificate and a certificate signing request, refer to Create a client certificate using your CA certificate.

> **Note**
>
> For security, the certificateOwnershipToken returned by CreateCertificateFromCsr expires after one hour. RegisterThing must be called before the certificateOwnershipToken expires. If the certificate created by CreateCertificateFromCsr hasn't been activated and attached to a policy or a thing by the time the token expires, the certificate is deleted. If the token expires, the device can call CreateCertificateFromCsr to generate a new certificate.

CreateCertificateFromCsr request

Publish a message with the `$aws/certificates/create-from-csr/`*`payload-format`* topic.

`payload-format`

The message payload format as `cbor` or `json`.

CreateCertificateFromCsr request payload

```
{
    "certificateSigningRequest": "string"
}
```

`certificateSigningRequest`

The CSR, in PEM format.

CreateCertificateFromCsr response

Subscribe to `$aws/certificates/create-from-csr/`*`payload-format`*`/accepted`.

`payload-format`

The message payload format as `cbor` or `json`.

CreateCertificateFromCsr response payload

```
{
    "certificateOwnershipToken": "string",
    "certificateId": "string",
    "certificatePem": "string"
}
```

`certificateOwnershipToken`

The token to prove ownership of the certificate during provisioning.

`certificateId`

The ID of the certificate. Certificate management operations only take a certificateId.

`certificatePem`

The certificate data, in PEM format.

CreateCertificateFromCsr error

To receive error responses, subscribe to `$aws/certificates/create-from-csr/`*`payload-format`*`/rejected`.

`payload-format`

The message payload format as `cbor` or `json`.

CreateCertificateFromCsr error payload

```
{
    "statusCode": int,
    "errorCode": "string",
    "errorMessage": "string"
}
```

`statusCode`

The status code.

`errorCode`

The error code.

`errorMessage`

The error message.

CreateKeysAndCertificate

Creates new keys and a certificate. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a `PENDING_ACTIVATION` status. When you call `RegisterThing` to provision a thing with this certificate, the certificate status changes to `ACTIVE` or `INACTIVE` as described in the template.

> **Note**
>
> For security, the `certificateOwnershipToken` returned by `CreateKeysAndCertificate` expires after one hour. `RegisterThing` must be called before the `certificateOwnershipToken` expires. If the certificate created by `CreateKeysAndCertificate` hasn't been activated and attached to a policy or a thing by the time the token expires, the certificate is deleted. If the token expires, the device can call `CreateKeysAndCertificate` to generate a new certificate.

CreateKeysAndCertificate request

Publish a message on `$aws/certificates/create/`*`payload-format`* with an empty message payload.

`payload-format`

The message payload format as `cbor` or `json`.

CreateKeysAndCertificate response

Subscribe to `$aws/certificates/create/`*`payload-format`*`/accepted`.

`payload-format`

The message payload format as `cbor` or `json`.

CreateKeysAndCertificate response

```
{
    "certificateId": "string",
    "certificatePem": "string",
    "privateKey": "string",
    "certificateOwnershipToken": "string"
}
```

`certificateId`

The certificate ID.

`certificatePem`

The certificate data, in PEM format.

`privateKey`

The private key.

`certificateOwnershipToken`

The token to prove ownership of the certificate during provisioning.

CreateKeysAndCertificate error

To receive error responses, subscribe to `$aws/certificates/create/`*`payload-format`*`/rejected`.

`payload-format`

The message payload format as `cbor` or `json`.

CreateKeysAndCertificate error payload

```
{
```

```
    "statusCode": int,
    "errorCode": "string",
    "errorMessage": "string"
}
```

`statusCode`

The status code.

`errorCode`

The error code.

`errorMessage`

The error message.

RegisterThing

Provisions a thing using a pre-defined template.

RegisterThing request

Publish a message on `$aws/provisioning-templates/`*`templateName`*`/provision/`*`payload-format`*.

`payload-format`

The message payload format as `cbor` or `json`.

`templateName`

The provisioning template name.

RegisterThing request payload

```
{
    "certificateOwnershipToken": "string",
    "parameters": {
        "string": "string",
        ...
    }
```

```
}
```

`certificateOwnershipToken`

The token to prove ownership of the certificate. AWS IoT generates the token when you create a certificate over MQTT.

`parameters`

Optional. Key-value pairs from the device that are used by the pre-provisioning hooks to evaluate the registration request.

RegisterThing response

Subscribe to `$aws/provisioning-templates/`*`templateName`*`/provision/`*`payload-format`*`/accepted`.

`payload-format`

The message payload format as `cbor` or `json`.

`templateName`

The provisioning template name.

RegisterThing response payload

```
{
    "deviceConfiguration": {
        "string": "string",
        ...
    },
    "thingName": "string"
}
```

`deviceConfiguration`

The device configuration defined in the template.

`thingName`

The name of the IoT thing created during provisioning.

RegisterThing error response

To receive error responses, subscribe to `$aws/provisioning-templates/`*`templateName`*`/provision/`*`payload-format`*`/rejected`.

`payload-format`

The message payload format as `cbor` or `json`.

`templateName`

The provisioning template name.

RegisterThing error response payload

```
{
    "statusCode": int,
    "errorCode": "string",
    "errorMessage": "string"
}
```

`statusCode`

The status code.

`errorCode`

The error code.

`errorMessage`

The error message.

Fleet indexing

You can use fleet indexing to index, search, and aggregate your devices' data from the following sources: AWS IoT registry, AWS IoT Device Shadow, AWS IoT connectivity, AWS IoT Device Management Software Package Catalog, and AWS IoT Device Defender violations. You can query a group of devices, and aggregate statistics on device records that are based on different combinations of device attributes, including state, connectivity, and device violations. With fleet indexing, you can organize, investigate, and troubleshoot your fleet of devices.

Fleet indexing provides the following capabilities.

Managing index updates

You can set up a fleet index to index updates for your thing groups, thing registries, device shadows, device connectivity, and device violations. When you activate fleet indexing, AWS IoT creates an index for your things or thing groups. `AWS_Things` is the index created for all of your things. `AWS_ThingGroups` is the index that contains all of your thing groups. After fleet indexing is active, you can run queries on your index. For example, you can find all devices that are handheld and have more than 70 percent battery life. AWS IoT updates the index continually with your latest data. For more information, see Managing fleet indexing.

Querying connectivity status for a specific device

This API provides low-latency, high-throughput access to the most recent device-specific connectivity information. For more information, see Device connectivity status.

Searching across data sources

You can create a query string based on a query language and use it to search across data sources. You also need to configure data sources in the fleet indexing setting so that the indexing configuration contains the data sources you want to search from. The query string describes the things that you want to find. You can create queries by using AWS managed fields, custom fields, and any attributes from your indexed data sources. For more information about data sources that support fleet indexing, see Managing thing indexing.

Querying for aggregate data

You can search your devices for aggregate data and return statistics, percentile, cardinality, or a list of things with search queries about particular fields. You can run aggregations on AWS managed fields or any attributes you configure as custom fields within fleet indexing settings. For more information about aggregation query, see Querying for aggregate data.

Monitoring aggregate data and creating alarms by using fleet metrics

You can use fleet metrics to send aggregate data to CloudWatch automatically, analyze trends, and create alarms to monitor the aggregate state of your fleet based on pre-defined thresholds. For more information about fleet metrics, see Fleet metrics.

Managing fleet indexing

Fleet indexing manages two types of indexes for you: thing indexing and thing group indexing.

Thing indexing

The index created for all of your things is called `AWS_Things`. Thing indexing supports the following data sources: AWS IoT registry data, AWS IoT Device Shadow data, AWS IoT connectivity data, and AWS IoT Device Defender violations data. By adding these data sources to your fleet indexing configuration, you can search for things, query for aggregate data, and create dynamic thing groups and fleet metrics based on your search queries.

Registry-AWS IoT provides a registry that helps you manage things. You can add the registry data to your fleet indexing configuration to search for devices based on the thing names, descriptions, and other registry attributes. For more information about the registry, see How to manage things with the registry.

Shadow-The AWS IoT Device Shadow service provides shadows that help you store your device state data. Thing indexing supports both classic unnamed shadows and named shadows. To index named shadows, activate your named shadow settings and specify your shadow names in thing indexing configuration. By default, you can add up to 10 shadow names per AWS account. To see how to increase the number of shadow names limit, see AWS IoT Device Management Quotas in the *AWS General Reference*.

To add named shadows for indexing:

- If you use the AWS IoT console, turn on **Thing indexing**, choose **Add named shadows**, and add your shadow names through **Named shadow selection**.
- If you use the AWS Command Line Interface (AWS CLI), set `namedShadowIndexingMode` to be `ON`, and specify shadow names in `IndexingFilter`. To see example CLI commands, see Manage thing indexing.

> **Important**
>
> July 20, 2022 is the General Availability (GA) release of the AWS IoT Device Management fleet indexing integration with AWS IoT Core named shadows and AWS IoT Device Defender detect violations. With this GA release, you can index specific named shadows by specifying shadow names. If you added your named shadows for indexing during this feature's public preview period from November 30, 2021 to July 19, 2022, we encourage you to reconfigure your fleet indexing settings and choose specific shadow names to reduce indexing cost and optimize performance.

For more information about shadows, see AWS IoT Device Shadow service.

Connectivity-Device connectivity data helps you identify the connection status of your devices. This connectivity data is driven by lifecycle events. When a client connects or disconnects, AWS IoT publishes lifecycle events with messages to MQTT topics. A connect or disconnect message can be a list of JSON elements that provide details of the connection status. For more information about device connectivity, see Lifecycle events.

Device Defender violations-AWS IoT Device Defender violations data helps identify anomalous device behaviors against the normal behaviors that you define in a Security Profile. A Security Profile contains a set of expected device behaviors. Each behavior uses a metric that specifies the normal behavior of your devices. For more information about Device Defender violations, see AWS IoT Device Defender detect.

For more information, see Managing thing indexing.

Thing group indexing

`AWS_ThingGroups` is the index that contains all of your thing groups. You can use this index to search for groups based on group name, description, attributes, and all parent group names.

For more information, see Managing thing group indexing.

Managed fields

Managed fields contain data associated with things, thing groups, device shadows, device connectivity, and Device Defender violations. AWS IoT defines the data type in managed fields. You specify the values of each managed field when you create an AWS IoT thing. For example, thing names, thing groups, and thing descriptions are all managed fields. Fleet indexing indexes managed fields based on the indexing mode that you specify. Managed fields can't be changed or appear in `customFields`. For more information, see Custom fields.

The following lists managed fields for thing indexing:

- Managed fields for the registry

```
"managedFields" : [
  {name:thingId, type:String},
  {name:thingName, type:String},
  {name:registry.version, type:Number},
  {name:registry.thingTypeName, type:String},
  {name:registry.thingGroupNames, type:String},
]
```

- Managed fields for classic unnamed shadows

```
"managedFields" : [
  {name:shadow.version, type:Number},
  {name:shadow.hasDelta, type:Boolean}
]
```

- Managed fields for named shadows

```
"managedFields" : [
  {name:shadow.name.shadowName.version, type:Number},
  {name:shadow.name.shadowName.hasDelta, type:Boolean}
]
```

- Managed fields for thing connectivity

```
"managedFields" : [
  {name:connectivity.timestamp, type:Number},
  {name:connectivity.version, type:Number},
  {name:connectivity.connected, type:Boolean},
  {name:connectivity.disconnectReason, type:String}
]
```

- Managed fields for Device Defender

```
"managedFields" : [
  {name:deviceDefender.violationCount, type:Number},
  {name:deviceDefender.securityprofile.behaviorname.metricName, type:String},
  {name:deviceDefender.securityprofile.behaviorname.lastViolationTime, type:Number},
  {name:deviceDefender.securityprofile.behaviorname.lastViolationValue, type:String},
  {name:deviceDefender.securityprofile.behaviorname.inViolation, type:Boolean}
]
```

- Managed fields for thing groups

```
"managedFields" : [
  {name:description, type:String},
  {name:parentGroupNames, type:String},
  {name:thingGroupId, type:String},
  {name:thingGroupName, type:String},
  {name:version, type:Number},
]
```

The following table lists managed fields that are not searchable.

Data source	Managed field that is unsearchable
Registry	`registry.version`
Unnamed shadows	`shadow.version`
Named shadows	`shadow.name.*.version`
Device Defender	`deviceDefender.version`

Data source	Managed field that is unsearchable
Thing groups	version

Custom fields

You can aggregate thing attributes, Device Shadow data, and Device Defender violations data by creating custom fields to index them. The `customFields` attribute is a list of field name and data type pairs. You can perform aggregation queries based on data type. The indexing mode that you choose affects fields can be specified in `customFields`. For example, if you specify the `REGISTRY` indexing mode, you can't specify a custom field from a thing shadow. You can use the update-indexing-configuration CLI command to create or update the custom fields (see an example command in Updating indexing configuration examples).

- **Custom field names**

Custom field names for thing and thing group attributes begin with `attributes.`, followed by the attribute name. If unnamed shadow indexing is on, things can have custom field names that begin with `shadow.desired` or `shadow.reported`, followed by the unnamed shadow data value name. If named shadow indexing is on, things can have custom field names that begin with `shadow.name.*.desired.` or `shadow.name.*.reported.`, followed by the named shadow data value. If Device Defender violations indexing is on, things can have custom field names that begin with `deviceDefender.`, followed by the Device Defender violations data value.

The attribute or data value name that follows the prefix can have only alphanumeric, - (hyphen), and _ (underscore) characters. It can't have any spaces.

If there' a type inconsistency between a custom field in your configuration and the value being indexed, fleet indexing ignores the inconsistent value for aggregation queries. CloudWatch Logs are helpful when troubleshooting aggregation query problems. For more information, see Troubleshooting aggregation queries for the fleet indexing service.

- **Custom field types**

Custom field types have the following supported values: `Number`, `String`, and `Boolean`.

Manage thing indexing

The index created for all of your things is `AWS_Things`. You can control what to index from the following data sources: AWS IoT registry data, AWS IoT Device Shadow data, AWS IoT connectivity data, and AWS IoT Device Defender violations data.

In this topic:

- Enabling thing indexing
- Describing a thing index
- Querying a thing index
- Restrictions and limitations
- Authorization

Enabling thing indexing

You use the update-indexing-configuration CLI command or the UpdateIndexingConfiguration API operation to create the `AWS_Things` index and control its configuration. By using the `--thing-indexing-configuration` (`thingIndexingConfiguration`) parameter, you control what kind of data (for example, registry, shadow, device connectivity data, and Device Defender violations data) is indexed.

The `--thing-indexing-configuration` parameter takes a string with the following structure:

```
{
  "thingIndexingMode": "OFF"|"REGISTRY"|"REGISTRY_AND_SHADOW",
  "thingConnectivityIndexingMode": "OFF"|"STATUS",
  "deviceDefenderIndexingMode": "OFF"|"VIOLATIONS",
  "namedShadowIndexingMode": "OFF"|"ON",
  "managedFields": [
    {
      "name": "string",
      "type": "Number"|"String"|"Boolean"
    },
    ...
  ],
  "customFields": [
    {
      "name": "string",
      "type": "Number"|"String"|"Boolean"
```

```
    },
    ...
  ],
  "filter": {
     "namedShadowNames": [ "string" ],
     "geoLocations": [
        {
            "name": "String",
            "order": "LonLat|LatLon"
        }
    ]
  }
}
```

Thing indexing modes

You can specify different thing indexing modes in your indexing configuration, depending on what data sources you want to index and search devices from:

- `thingIndexingMode`: Controls if registry or shadow is indexed. When `thingIndexingMode` is set to be `OFF`, thing indexing is disabled.
- `thingConnectivityIndexingMode`: Specifies if thing connectivity data is indexed.
- `deviceDefenderIndexingMode`: Specifies if Device Defender violations data is indexed.
- `namedShadowIndexingMode`: Specifies if named shadow data is indexed. To select named shadows to add to your fleet indexing configuration, set `namedShadowIndexingMode` to be `ON` and specify your named shadow names in [`filter`](#).

The table below shows the valid values for each indexing mode and the data source that's indexed for each value.

Attribute	Valid values	Registry	Shadow	Connectivity	DD violations	Named shadow
`thingIndexingMode`	OFF					
	REGISTRY	✓				

Attribute	Valid values	Registry	Shadow	Connectivity	DD violations	Named shadow
	REGISTRY_AND_SHADOW	✓	✓			
`thingConnectivityIndexingMode`	*Not specified.*					
	OFF					
	STATUS			✓		
`deviceDefenderIndexingMode`	*Not specified.*					
	OFF					
	VIOLATIONS				✓	
`namedShadowIndexingMode`	*Not specified.*					
	OFF					
	ON					✓

Managed fields and custom fields

Managed fields

Managed fields contain data associated with things, thing groups, device shadows, device connectivity, and Device Defender violations. AWS IoT defines the data type in managed fields. You specify the values of each managed field when you create an AWS IoT thing. For example, thing names, thing groups, and thing descriptions are all managed fields. Fleet indexing indexes managed fields based on the indexing mode that you specify. Managed fields can't be changed or appear in `customFields`.

Custom fields

You can aggregate attributes, Device Shadow data, and Device Defender violations data by creating custom fields to index them. The `customFields` attribute is a list of field name and data type pairs. You can perform aggregation queries based on data type. The indexing mode that you choose affects fields can be specified in `customFields`. For example, if you specify the `REGISTRY` indexing mode, you can't specify a custom field from a thing shadow. You can use the [update-indexing-configuration](#) CLI command to create or update the custom fields (see an example command in [Updating indexing configuration examples](#)). For more information, see [Custom fields](#).

Indexing filter

Indexing filter provides additional selections for named shadows and geolocation data.

`namedShadowNames`

To add named shadows to your fleet indexing configuration, set `namedShadowIndexingMode` to be `ON` and specify your named shadow names in `namedShadowNames` filter.

Example

```
"filter": {
     "namedShadowNames": [ "namedShadow1", "namedShadow2" ]
}
```

`geoLocations`

To add geolocation data to your fleet indexing configuration:

- If your geolocation data is stored in a classic (unnamed) shadow, set `thingIndexingMode` to be REGISTRY_AND_SHADOW, and specify your geolocation data in `geoLocations` filter.

  The example filter below specifies a geoLocation object in a classic (unnamed) shadow:

  ```
  "filter": {
       "geoLocations": [
          {
              "name": "shadow.reported.location",
              "order": "LonLat"
          }
       ]
    }
  ```

- If your geolocation data is stored in a named shadow, set `namedShadowIndexingMode` to be ON, add the shadow name in `namedShadowNames` filter, and specify your geolocation data in `geoLocations` filter.

  The example filter below specifies a geoLocation object in a named shadow (`nameShadow1`):

```
"filter": {
     "namedShadowNames": [ "namedShadow1" ],
     "geoLocations": [
        {
            "name": "shadow.name.namedShadow1.reported.location",
            "order": "LonLat"
        }
     ]
  }
```

For more information, see [IndexingFilter](#) from *AWS IoT API Reference*.

Updating indexing configuration examples

To update your indexing configuration, use the AWS IoT **update-indexing-configuration** CLI command . The following examples show how to use **update-indexing-configuration**.

Short syntax:

```
aws iot update-indexing-configuration --thing-indexing-configuration \
'thingIndexingMode=REGISTRY_AND_SHADOW, deviceDefenderIndexingMode=VIOLATIONS,
namedShadowIndexingMode=ON,filter={namedShadowNames=[thing1shadow]},
 thingConnectivityIndexingMode=STATUS,
customFields=[{name=attributes.version,type=Number},
{name=shadow.name.thing1shadow.desired.DefaultDesired, type=String},
 {name=shadow.desired.power, type=Boolean},
{name=deviceDefender.securityProfile1.NUMBER_VALUE_BEHAVIOR.lastViolationValue.number,
 type=Number}]'
```

JSON syntax:

```
aws iot update-indexing-configuration --cli-input-json \ '{
          "thingIndexingConfiguration": { "thingIndexingMode": "REGISTRY_AND_SHADOW",
          "thingConnectivityIndexingMode": "STATUS",
          "deviceDefenderIndexingMode": "VIOLATIONS",
```

```
        "namedShadowIndexingMode": "ON",
        "filter": { "namedShadowNames": ["thing1shadow"]},
        "customFields": [ { "name": "shadow.desired.power", "type": "Boolean" },
        {"name": "attributes.version", "type": "Number"},
        {"name": "shadow.name.thing1shadow.desired.DefaultDesired", "type":
"String"},
        {"name":
"deviceDefender.securityProfile1.NUMBER_VALUE_BEHAVIOR.lastViolationValue.number",
"type": Number} ] } }'
```

This command doesn't produce any output.

To check the thing index status, run the `describe-index` CLI command:

```
aws iot describe-index --index-name "AWS_Things"
```

The output of the `describe-index` command looks like the following:

```
{
    "indexName": "AWS_Things",
    "indexStatus": "ACTIVE",
    "schema": "MULTI_INDEXING_MODE"
}
```

Note

It can take a moment for fleet indexing to update the fleet index. We recommend waiting until the `indexStatus` shows ACTIVE before using it. You can have different values in the schema field depending on what data sources you've configured. For more information, see [Describing a thing index](#).

To get your thing indexing configuration details, run the `get-indexing-configuration` CLI command:

```
aws iot get-indexing-configuration
```

The output of the `get-indexing-configuration` command looks like the following:

```
{
```

```
    "thingIndexingConfiguration": {
        "thingIndexingMode": "REGISTRY_AND_SHADOW",
        "thingConnectivityIndexingMode": "STATUS",
        "deviceDefenderIndexingMode": "VIOLATIONS",
        "namedShadowIndexingMode": "ON",
        "managedFields": [
            {
                "name": "connectivity.disconnectReason",
                "type": "String"
            },
            {
                "name": "registry.version",
                "type": "Number"
            },
            {
                "name": "thingName",
                "type": "String"
            },
            {
                "name": "deviceDefender.violationCount",
                "type": "Number"
            },
            {
                "name": "shadow.hasDelta",
                "type": "Boolean"
            },
            {
                "name": "shadow.name.*.version",
                "type": "Number"
            },
            {
                "name": "shadow.version",
                "type": "Number"
            },
            {
                "name": "connectivity.version",
                "type": "Number"
            },
            {
                "name": "connectivity.timestamp",
                "type": "Number"
            },
            {
                "name": "shadow.name.*.hasDelta",
```

```
                "type": "Boolean"
            },
            {
                "name": "registry.thingTypeName",
                "type": "String"
            },
            {
                "name": "thingId",
                "type": "String"
            },
            {
                "name": "connectivity.connected",
                "type": "Boolean"
            },
            {
                "name": "registry.thingGroupNames",
                "type": "String"
            }
        ],
        "customFields": [
            {
                "name": "shadow.name.thing1shadow.desired.DefaultDesired",
                "type": "String"
            },

            {
                "name": 
 "deviceDefender.securityProfile1.NUMBER_VALUE_BEHAVIOR.lastViolationValue.number",
                "type": "Number"
            },
            {
                "name": "shadow.desired.power",
                "type": "Boolean"
            },
            {
                "name": "attributes.version",
                "type": "Number"
            }
        ],
        "filter": {
              "namedShadowNames": [
                  "thing1shadow"
              ]
          }
```

```
    },
    "thingGroupIndexingConfiguration": {
        "thingGroupIndexingMode": "OFF"
    }
}
```

To update the custom fields, you can run the `update-indexing-configuration` command. An example is as follows:

```
aws iot update-indexing-configuration --thing-indexing-configuration

 'thingIndexingMode=REGISTRY_AND_SHADOW,customFields=[{name=attributes.version,type=Number},
{name=attributes.color,type=String},{name=shadow.desired.power,type=Boolean},
{name=shadow.desired.intensity,type=Number}]'
```

This command added `shadow.desired.intensity` to the indexing configuration.

> **Note**
>
> Updating the custom field indexing configuration overwrites all existing custom fields. Make sure to specify all custom fields when calling **update-indexing-configuration**.

After the index is rebuilt, you can use an aggregation query on the newly added fields, search registry data, shadow data, and thing connectivity status data.

When changing the indexing mode, make sure all of your custom fields are valid by using the new indexing mode. For example, if you start off using `REGISTRY_AND_SHADOW` mode with a custom field called `shadow.desired.temperature`, you must delete the `shadow.desired.temperature` custom field before changing the indexing mode to `REGISTRY`. If your indexing configuration contains custom fields that aren't indexed by the indexing mode, the update fails.

Describing a thing index

The following command shows you how to use the **describe-index** CLI command to retrieve the current status of the thing index.

```
aws iot describe-index --index-name "AWS_Things"
```

The response of the command can look like the following:

```
{
    "indexName": "AWS_Things",
    "indexStatus": "BUILDING",
    "schema": "REGISTRY_AND_SHADOW_AND_CONNECTIVITY_STATUS"
}
```

The first time that you fleet indexing, AWS IoT builds your index. When `indexStatus` is in the `BUILDING` state, you can't query the index. The `schema` for the things index indicates which type of data (`REGISTRY_AND_SHADOW_AND_CONNECTIVITY_STATUS`) is indexed.

Changing the configuration of your index causes the index to be rebuilt. During this process, the `indexStatus` is `REBUILDING`. You can run queries on data in the things index while it's being rebuilt. For example, if you change the index configuration from `REGISTRY` to `REGISTRY_AND_SHADOW` while the index is being rebuilt, you can query registry data, including the latest updates. However, you can't query the shadow data until the rebuild is complete. The amount of time it takes to build or rebuild the index depends on the amount of data.

You can see different values in the schema field depending on the data sources that you've configured. The following table shows the different schema values and the corresponding descriptions:

Schema	Description
OFF	No data sources are configured or indexed.
REGISTRY	Registry data is indexed.
REGISTRY_AND_SHADOW	Registry data and unnamed (classic) shadow data are indexed.
REGISTRY_AND_CONNECTIVITY	Registry data and connectivity data are indexed.
REGISTRY_AND_SHADOW_AND_CON NECTIVITY_STATUS	Registry data, unnamed (classic) shadow data, and connectivity data are indexed.
MULTI_INDEXING_MODE	Named shadow or Device Defender violation s data is indexed, in addition to registry,

Schema	Description
	unnamed (classic) shadow or connectivity data.

Querying a thing index

Use the **search-index** CLI command to query data in the index.

```
aws iot search-index --index-name "AWS_Things" --query-string
          "thingName:mything*"
```


```
{
    "things":[{
         "thingName":"mything1",
         "thingGroupNames":[
            "mygroup1"
         ],
         "thingId":"a4b9f759-b0f2-4857-8a4b-967745ed9f4e",
         "attributes":{
            "attribute1":"abc"
         },
         "connectivity": {
            "connected":false,
            "timestamp":1556649874716,
            "disconnectReason": "CONNECTION_LOST"
         }
    },
    {
        "thingName":"mything2",
        "thingTypeName":"MyThingType",
        "thingGroupNames":[
            "mygroup1",
            "mygroup2"
        ],
        "thingId":"01014ef9-e97e-44c6-985a-d0b06924f2af",
        "attributes":{
            "model":"1.2",
            "country":"usa"
        },
        "shadow":{
```

```
            "desired":{
                "location":"new york",
                "myvalues":[3, 4, 5]
            },
            "reported":{
                "location":"new york",
                "myvalues":[1, 2, 3],
                "stats":{
                    "battery":78
                }
            },
            "metadata":{
                 "desired":{
                       "location":{
                            "timestamp":123456789
                        },
                       "myvalues":{
                             "timestamp":123456789
                       }
                  },
                  "reported":{
                        "location":{
                             "timestamp":34535454
                         },
                        "myvalues":{
                             "timestamp":34535454
                        },
                        "stats":{
                             "battery":{
                                   "timestamp":34535454
                             }
                        }
                 }
            },
            "version":10,
            "timestamp":34535454
        },
        "connectivity": {
            "connected":true,
            "timestamp":1556649855046
        }
    }],
    "nextToken":"AQFCuvk7zZ3D9pOYMbFCeHbdZ+h=G"
```

```
}
```

In the JSON response, "connectivity" (as enabled by the thingConnectivityIndexingMode=STATUS setting) provides a Boolean value, a timestamp, and a disconnectReason that indicates whether the device is connected to AWS IoT Core. The device "mything1" disconnected (false) at POSIX time 1556649874716 due to CONNECTION_LOST. For more information about disconnect reasons, see Lifecycle events.

```
"connectivity": {
    "connected":false,
    "timestamp":1556649874716,
    "disconnectReason": "CONNECTION_LOST"
}
```

The device "mything2" connected (true) at POSIX time 1556649855046:

```
"connectivity": {
    "connected":true,
    "timestamp":1556649855046
}
```

Timestamps are given in milliseconds since epoch, so 1556649855046 represents 6:44:15.046 PM on Tuesday, April 30, 2019 (UTC).

> **Important**
>
> If a device has been disconnected for approximately an hour, the "timestamp" value and the "disconnectReason" value of the connectivity status might be missing.

Restrictions and limitations

These are the restrictions and limitations for AWS_Things.

Shadow fields with complex types

A shadow field is indexed only if the value of the field is a simple type, such as a JSON object that doesn't contain an array, or an array that consists entirely of simple types. Simple type means a string, number, or one of the literals true or false. For example, given the following shadow state, the value of field "palette" isn't indexed because it's an array that contains items of complex types. The value of field `"colors"` is indexed because each value in the array is a string.

```
{
    "state": {
        "reported": {
            "switched": "ON",
            "colors": [ "RED", "GREEN", "BLUE" ],
            "palette": [
                {
                    "name": "RED",
                    "intensity": 124
                },
                {
                    "name": "GREEN",
                    "intensity": 68
                },
                {
                    "name": "BLUE",
                    "intensity": 201
                }
            ]
        }
    }
}
```

Nested shadow field names

The names of nested shadow fields are stored as a period (.) delimited string. For example, given a shadow document:

```
{
  "state": {
    "desired": {
      "one": {
        "two": {
          "three": "v2"
        }
      }
    }
  }
}
```

The name of field `three` is stored as `desired.one.two.three`. If you also have a shadow document, it's stored like this:

```
{
  "state": {
    "desired": {
      "one.two.three": "v2"
    }
  }
}
```

Both match a query for `shadow.desired.one.two.three:v2`. As a best practice, don't use periods in shadow field names.

Shadow metadata

A field in a shadow's metadata section is indexed, but only if the corresponding field in the shadow's `"state"` section is indexed. (In the previous example, the `"palette"` field in the shadow's metadata section isn't indexed either.)

Unregistered devices

Fleet indexing indexes the connectivity status for a device whose connection `clientId` is the same as the `thingName` of a registered thing in [Registry](#).

Unregistered shadows

If you use [UpdateThingShadow](#) to create a shadow using a thing name that hasn't been registered in your AWS IoT account, fields in this shadow aren't indexed. This applies to both classic unnamed shadow and named shadow.

Numeric values

If any registry or shadow data is recognized by the service as a numeric value, it's indexed as such. You can form queries involving ranges and comparison operators on numeric values (for example, `"attribute.foo<5"` or `"shadow.reported.foo:[75 TO 80]"`). To be recognized as numeric, the value of the data must be a valid, literal type JSON number. The value can be an integer in the range -2^53...2^53-1, a double-precision floating point with optional exponential notation, or part of an array that contains only these values.

Null values

Null values aren't indexed.

Maximum values

The maximum number of custom fields for aggregation queries is 5.

The maximum number of requested percentiles for aggregation queries is 100.

Authorization

You can specify the things index as an Amazon Resource Name (ARN) in an AWS IoT policy action, as follows.

Action	Resource
`iot:SearchIndex`	An index ARN (for example, `arn:aws:iot:` *`your-aws-region your-aws-account`* `:index/AWS_Things`).
`iot:DescribeIndex`	An index ARN (for example, `arn:aws:iot:` *`your-aws-region`* `:index/AWS_Things`).

> **Note**
>
> If you have permissions to query the fleet index, you can access the data of things across the entire fleet.

Manage thing group indexing

`AWS_ThingGroups` is the index that contains all of your thing groups. You can use this index to search for groups based on group name, description, attributes, and all parent group names.

Enabling thing group indexing

You can use the `thing-group-indexing-configuration` setting in the UpdateIndexingConfiguration API to create the `AWS_ThingGroups` index and control its configuration. You can use the GetIndexingConfiguration API to retrieve the current indexing configuration.

To update the thing group indexing configurations, run the **update-indexing-configuration** CLI command:

```
aws iot update-indexing-configuration --thing-group-indexing-configuration
 thingGroupIndexingMode=ON
```

You can also update configurations for both thing and thing group indexing in a single command, as follows:

```
aws iot update-indexing-configuration --thing-indexing-configuration
 thingIndexingMode=REGISTRY --thing-group-indexing-configuration
 thingGroupIndexingMode=ON
```

The following are valid values for `thingGroupIndexingMode`.

OFF

No indexing/delete index.

ON

Create or configure the `AWS_ThingGroups` index.

To retrieve the current thing and thing group indexing configurations, run the **get-indexing-configuration** CLI command:

```
aws iot get-indexing-configuration
```

The response of the command looks like the following:

```
{
   "thingGroupIndexingConfiguration": {
        "thingGroupIndexingMode": "ON"
    }
}
```

Describing group indexes

To retrieve the current status of the `AWS_ThingGroups` index, use the **describe-index** CLI command:

```
aws iot describe-index --index-name "AWS_ThingGroups"
```

The response of the command looks like the following:

```
{
   "indexStatus": "ACTIVE",
   "indexName": "AWS_ThingGroups",
   "schema": "THING_GROUPS"
}
```

AWS IoT builds your index the first time that you indexing. You can't query the index if the `indexStatus` is `BUILDING`.

Querying a thing group index

To query data in the index, use the **search-index** CLI command:

```
aws iot search-index --index-name "AWS_ThingGroups" --query-string
 "thingGroupName:mythinggroup*"
```

Authorization

You can specify the thing groups index as a resource ARN in an AWS IoT policy action, as follows.

Action	Resource
`iot:SearchIndex`	An index ARN (for example, `arn:aws:iot:` *your-aws-region* `:index/AWS_ThingGroups`).
`iot:DescribeIndex`	An index ARN (for example, `arn:aws:iot:` *your-aws-region* `:index/AWS_ThingGroups`).

Device connectivity status queries

AWS IoT Fleet Indexing supports individual device connectivity querying, allowing you to efficiently retrieve connectivity status and related metadata for specific devices. This feature complements existing fleet-wide indexing and querying capabilities.

How it works

Device connectivity query support can be used for optimized single-device connectivity status retrieval. This API provides low-latency, high-throughput access to the most recent device-specific connectivity information. Once you enable connectivity indexing you will have access to this query API which will be charged as standard queries. For more information, see [AWS IoT Device Management pricing]

Features

With device connectivity query support, you can:

1. Query the current connectivity state (connected or disconnected) for a given device using its `thingName`.
2. Retrieve additional connectivity metadata, including:
   a. Disconnect reason
   b. Timestamps for the most recent connect or disconnect event.

> **Note**
>
> Fleet indexing indexes the connectivity status for a device whose connection `clientId` is the same as the `thingName` of a registered thing in [Registry.]

Benefits

1. **Low latency:** Reflects the most recent device connectivity state and offers low latency to reflect connection state changes from IoT Core. IoT Core determines a device as disconnected either as soon as it receives a disconnect request from the device or in case of a device disconnecting without sending a disconnect request. IoT core will wait 1.5x of the configured keep-alive time

before the client is determined to be disconnected. The Connectivity status API will reflect these changes typically under one second after IoT Core determines a device's connected state change.

2. **High throughput:** Supports 350 Transactions Per Second (TPS) by default, and can be adjustable to higher upon request.
3. **Data retention:** Stores event data indefinitely when Fleet Indexing (FI) ConnectivityIndexing mode is enabled and the thing is not deleted. If you disable Connectivity Indexing, the records will not be retained.

> **Note**
>
> If connectivity status indexing was enabled before the launch of this API, Fleet Indexing begins tracking connectivity status changes after the API launch and reflects the updated status based on those changes.

Prerequisites

To use device connectivity query support:

1. Set up an AWS account
2. Onboard and register devices to AWS IoT Core in your preferred region
3. Enable Fleet Indexing with Connectivity indexing

> **Note**
>
> No additional setup is required if you already have connectivity indexing enabled

For detailed setup instructions, refer to the AWS IoT Developer Guide

Examples

```
aws iot get-thing-connectivity-data --thing-name myThingName
```

```
{
   "connected": true,
   "disconnectReason": "NONE",
   "thingName": "myThingName",
   "timestamp": "2024-12-19T10:00:00.000000-08:00"
}
```

- `thingName`: The name of the device as indicated by the request. This also matches the clientId used to connect to AWS IoT Core.
- `disconnectReason`: Reason for disconnect. Will be NONE for a connected device.
- `connected`: The boolean value true indicating this device is currently connected.
- `timestamp`: The timestamp representing the device's most recent disconnect in milliseconds.

```
aws iot get-thing-connectivity-data --thing-name myThingName
```

```
{
   "connected": false,
   "disconnectReason": "CLIENT_INITIATED_DISCONNECT",
   "thingName": "myThingName",
   "timestamp": "2024-12-19T10:30:00.000000-08:00"
}
```

- `thingName`: The name of the device as indicated by the request. This also matches the clientId used to connect to AWS IoT Core.
- `disconnectReason`: Reason for disconnect is CLIENT_INITIATED_DISCONNECT indicating the client indicated to AWS IoT Core that it would disconnect.
- `connected`: The boolean value false indicating this device is currently disconnected.
- `timestamp`: The timestamp representing the device's most recent disconnect in milliseconds.

```
aws iot get-thing-connectivity-data --thing-name neverConnectedThing
```

```
{
   "connected": false,
   "disconnectReason": "UNKNOWN",
```

```
    "thingName": "neverConnectedThing"
}
```

- `thingName`: The name of the device as indicated by the request. This also matches the clientId used to connect to AWS IoT Core.
- `disconnectReason`: Reason for disconnect. Will be "UNKNOWN" for a device which has never been connected or for which Fleet Indexing does not have the last disconnect reason stored.
- `connected`: The boolean value false indicating this device is currently disconnected.
- `timestamp`: The timestamp is not returned for a device which has never been connected or for which Fleet Indexing does not have the last timestamp stored.

Querying for aggregate data

AWS IoT provides four APIs (`GetStatistics`, `GetCardinality`, `GetPercentiles`, and `GetBucketsAggregation`) that allow you to search your device fleet for aggregate data.

> **Note**
>
> For issues with missing or unexpected values for the aggregation APIs, read Fleet indexing troubleshooting guide.

GetStatistics

The GetStatistics API and the **get-statistics** CLI command return the count, average, sum, minimum, maximum, sum of squares, variance, and standard deviation for the specified aggregated field.

The **get-statistics** CLI command takes the following parameters:

`index-name`

The name of the index to search. The default value is `AWS_Things`.

`query-string`

The query used to search the index. You can specify `"*"` to get the count of all indexed things in your AWS account.

`aggregationField`

(Optional)The field to aggregate. This field must be a managed or custom field defined when you call **update-indexing-configuration**. If you don't specify an aggregation field, `registry.version` is used as the aggregation field.

`query-version`

The version of the query to use. The default value is `2017-09-30`.

The type of aggregation field can affect the statistics returned.

GetStatistics with string values

If you aggregate on a string field, calling `GetStatistics` returns a count of devices that have attributes that match the query. For example:

```
aws iot get-statistics --aggregation-field 'attributes.stringAttribute'
            --query-string '*'
```

This command returns the number of devices that contain an attribute named `stringAttribute`:

```
{
  "statistics": {
    "count": 3
  }
}
```

GetStatistics with Boolean values

When you call `GetStatistics` with a Boolean aggregation field:

- AVERAGE is the percentage of devices that match the query.
- MINIMUM is 0 or 1 according to the following rules:
  - If all the values for the aggregation field are `false`, MINIMUM is 0.
  - If all the values for the aggregation field are `true`, MINIMUM is 1.
  - If the values for the aggregation field are a mixture of `false` and `true`, MINIMUM is 0.
- MAXIMUM is 0 or 1 according to the following rules:
  - If all the values for the aggregation field are `false`, MAXIMUM is 0.

  - If all the values for the aggregation field are `true`, MAXIMUM is 1.
  - If the values for the aggregation field are a mixture of `false` and `true`, MAXIMUM is 1.
- SUM is the sum of the integer equivalent of the Boolean values.
- COUNT is the count of things that match the query string criteria and contain a valid aggregation field value.

GetStatistics with numerical values

When you call `GetStatistics` and specify an aggregation field of type `Number`, `GetStatistics` returns the following values:

count

The count of things that match the query string criteria and contain a valid aggregation field value.

average

The average of the numerical values that match the query.

sum

The sum of the numerical values that match the query.

minimum

The smallest of the numerical values that match the query.

maximum

The largest of the numerical values that match the query.

sumOfSquares

The sum of the squares of the numerical values that match the query.

variance

The variance of the numerical values that match the query. The variance of a set of values is the average of the squares of the differences of each value from the average value of the set.

stdDeviation

The standard deviation of the numerical values that match the query. The standard deviation of a set of values is a measure of how spread out the values are.

The following example shows how to call **get-statistics** with a numerical custom field.

```
aws iot get-statistics --aggregation-field 'attributes.numericAttribute2'
            --query-string '*'
```

```
{
  "statistics": {
    "count": 3,
    "average": 33.333333333333336,
    "sum": 100.0,
    "minimum": -125.0,
    "maximum": 150.0,
    "sumOfSquares": 43750.0,
    "variance": 13472.22222222222,
    "stdDeviation": 116.06990230986766
  }
}
```

For numerical aggregation fields, if the field values exceed the maximum double value, the statistics values are empty.

GetCardinality

The GetCardinality API and the **get-cardinality** CLI command return the approximate count of unique values that match the query. For example, you might want to find the number of devices with battery levels at less than 50 percent:

```
aws iot get-cardinality --index-name AWS_Things --query-string "batterylevel
          > 50" --aggregation-field "shadow.reported.batterylevel"
```

This command returns the number of things with battery levels at more than 50 percent:

```
{
    "cardinality": 100
}
```

`cardinality` is always returned by **get-cardinality** even if there are no matching fields. For example:

```
aws iot get-cardinality --query-string "thingName:Non-existent*"
```

```
          --aggregation-field "attributes.customField_STR"
```

```
{
    "cardinality": 0
}
```

The **get-cardinality** CLI command takes the following parameters:

`index-name`

The name of the index to search. The default value is `AWS_Things`.

`query-string`

The query used to search the index. You can specify `"*"` to get the count of all indexed things in your AWS account.

`aggregationField`

The field to aggregate.

`query-version`

The version of the query to use. The default value is `2017-09-30`.

GetPercentiles

The [GetPercentiles](https://docs.aws.amazon.com/iot/latest/apireference/API_GetPercentiles.html) API and the **get-percentiles** CLI command groups the aggregated values that match the query into percentile groupings. The default percentile groupings are: 1,5,25,50,75,95,99, although you can specify your own when you call `GetPercentiles`. This function returns a value for each percentile group specified (or the default percentile groupings). The percentile group "1" contains the aggregated field value that occurs in approximately one percent of the values that match the query. The percentile group "5" contains the aggregated field value that occurs in approximately five percent of the values that match the query, and so on. The result is an approximation, the more values that match the query, the more accurate the percentile values.

The following example shows how to call the **get-percentiles** CLI command.

```
aws iot get-percentiles --query-string "thingName:*" --aggregation-field
```

```
        "attributes.customField_NUM" --percents 10 20 30 40 50 60 70 80 90 99
```


```
{
    "percentiles": [
        {
            "value": 3.0,
            "percent": 80.0
        },
        {
            "value": 2.5999999999999996,
            "percent": 70.0
        },
        {
            "value": 3.0,
            "percent": 90.0
        },
        {
            "value": 2.0,
            "percent": 50.0
        },
        {
            "value": 2.0,
            "percent": 60.0
        },
        {
            "value": 1.0,
            "percent": 10.0
        },
        {
            "value": 2.0,
            "percent": 40.0
        },
        {
            "value": 1.0,
            "percent": 20.0
        },
        {
            "value": 1.4,
            "percent": 30.0
        },
        {
            "value": 3.0,
            "percent": 99.0
```

```
        }
    ]
}
```

The following command shows the output returned from **get-percentiles** when there are no matching documents.

```
aws iot get-percentiles --query-string "thingName:Non-existent*"
          --aggregation-field "attributes.customField_NUM"
```


```
{
    "percentiles": []
}
```


The **get-percentile** CLI command takes the following parameters:

`index-name`

The name of the index to search. The default value is `AWS_Things`.

`query-string`

The query used to search the index. You can specify `"*"` to get the count of all indexed things in your AWS account.

`aggregationField`

The field to aggregate, which must be of `Number` type.

`query-version`

The version of the query to use. The default value is `2017-09-30`.

`percents`

(Optional)You can use this parameter to specify custom percentile groupings.

GetBucketsAggregation

The GetBucketsAggregation API and the **get-buckets-aggregation** CLI command return a list of buckets and the total number of things that fit the query string criteria.

The following example shows how to call the get-buckets-aggregation CLI command.

```
aws iot get-buckets-aggregation --query-string '*' --index-name AWS_Things --
aggregation-field 'shadow.reported.batterylevelpercent' --buckets-aggregation-type
 'termsAggregation={maxBuckets=5}'
```

This command returns the following:


```
{
    "totalCount": 20,
    "buckets": [
        {
            "keyValue": "100",
            "count": 12
        },
        {
            "keyValue": "90",
            "count": 5
        },
        {
            "keyValue": "75",
            "count": 3
        }
    ]
}
```


The get-buckets-aggregation CLI command takes the following parameters:

`index-name`

The name of the index to search. The default value is `AWS_Things`.

`query-string`

The query used to search the index. You can specify `"*"` to get the count of all indexed things in your AWS account.

`aggregation-field`

The field to aggregate.

`buckets-aggregation-type`

The basic control of the response shape and the bucket aggregation type to perform.

Authorization

You can specify the thing groups index as a resource ARN in an AWS IoT policy action, as follows.

Action	Resource
`iot:GetStatistics`	An index ARN (for example, `arn:aws:iot: your-aws-region :index/AWS_Things` or `arn:aws:iot: your-aws-region :index/AWS_ThingGroups`).

Query syntax

In fleet indexing, you use a query syntax to specify queries.

Supported features

The query syntax supports the following features:

- Terms and phrases
- Searching fields
- Prefix search
- Range search
- Boolean operators `AND`, `OR`, `NOT`, and `–`. The hyphen is used to exclude something from search results (for example, `thingName:(tv* AND -plasma)`).
- Grouping
- Field grouping
- Escaping special characters (such as with \)
- Leading wildcard use is limited to 1 query term per query. For example, you cannot search for `thingName:*my` and `thingGroupNames:*echo` in the same query. Queries that include a leading wildcard have a max query length of 100 characters.

> **Note**
>
> Accounts with a Fleet Indexing query API limit exceeding 30 requests per second are subject to throttling when using leading-wildcard query terms. This 30 requests-per-second limit applies at the account level across all Fleet Indexing query APIs.

Unsupported features

The query syntax doesn't support the following features:

- Regular expressions
- Boosting
- Ranking
- Fuzzy searches
- Proximity search
- Sorting
- Aggregation
- Special characters: `` ` ``, `@`, `#`, `%`, `\`, `/`, `'`, `;`, and `,`. Note that `,` is only supported in geoqueries.

Notes

A few things to note about the query language:

- The default operator is AND. A query for `"thingName:abc thingType:xyz"` is equivalent to `"thingName:abc AND thingType:xyz"`.
- If a field isn't specified, AWS IoT searches for the term in all the registry, Device Shadow, and Device Defender fields.
- All field names are case sensitive.
- Search is case insensitive. Words are separated by white-space characters as defined by Java's `Character.isWhitespace(int)`.
- Indexing of Device Shadow data (unnamed shadows and named shadows) includes reported, desired, delta, and metadata sections.
- Device shadow and registry versions aren't searchable, but are present in the response.

- The maximum number of terms in a query is twelve.
- The special character `,` is only supported in geoqueries.

Example thing queries

Specify queries in a query string using a query syntax. The queries are passed to the [SearchIndex](https://docs.aws.amazon.com/iot/latest/apireference/API_SearchIndex.html) API. The following table lists some example query strings.

Query string	Result
`abc`	Queries for "abc" in any registry, shadow (classic unnamed shadow and named shadow), or Device Defender violations field.
`thingName:myThingName`	Queries for a thing with name "myThingName".
`thingName:my*`	Queries for things with names that begin with "my".
`thingName:*my`	Queries for things with names that end in "my". Leading wildcard use is limited to 1 query term per query. Queries that include a leading wildcard have a max query length of 100 characters.
`thingName:*my*`	Queries for things that contain the substring "my". Leading wildcard use is limited to 1 query term per query. Queries that include a leading wildcard have a max query length of 100 characters.
`thingName:ab?`	Queries for things with names that have "ab" plus one additional character (for example, "aba", "abb", "abc", and so on.
`thingTypeName:aa`	Queries for things that are associated with type "aa".
`thingGroupNames:a`	Queries for things with a parent thing group or billing group name "a".

Query string	Result
`thingGroupNames:a*`	Queries for things with a parent thing group or billing group name matching the pattern "a*".
`attributes.myAttribute:75`	Queries for things with an attribute named "myAttribute" that has the value 75.
`attributes.myAttribute:[75 TO 80]`	Queries for things with an attribute named "myAttribute" that has a value that falls within a numeric range (75–80, inclusive).
`attributes.myAttribute:{75 TO 80]`	Queries for things with an attribute named "myAttribute" that has a value that falls within the numeric range (>75 and <=80).
`attributes.serialNumber:["abcd" TO "abcf"]`	Queries for things with an attribute named "serialNumber" that has a value within an alphanumeric string range. This query returns things with a "serialNumber" attribute with values "abcd", "abce", or "abcf".
`attributes.myAttribute:i*t`	Queries for things with an attribute named "myAttribute" where the value is 'i', followed by any number of characters, followed by 't'.
`attributes.attr1:abc AND attributes.attr2<5 NOT attributes.attr3>10`	Queries for things that combine terms using Boolean expressions. This query returns things that have an attribute named "attr1" with a value "abc", an attribute named "attr2" that's less than 5, and an attribute named "attr3" that' not greater than 10.
`shadow.hasDelta:true`	Queries for things with an unnamed shadow that has a delta element.
`NOT attributes.model:legacy`	Queries for things where the attribute named "model" is not "legacy".

Query string	Result
`shadow.reported.stats.battery:{70 TO 100} (v2 OR v3) NOT attributes.model:legacy`	Queries for things with the following: • The thing's shadow `stats.battery` attribute has a value between 70 and 100. • The text "v2" or "v3" occurs in a thing's name, type name, or attribute values. • The thing's `model` attribute is not set to "legacy".
`shadow.reported.myvalues:2`	Queries for things where the `myvalues` array in the shadow's reported section contains a value of 2.
`shadow.reported.location:* NOT shadow.desired.stats.battery:*`	Queries for things with the following: • The `location` attribute exists in the shadow's `reported` section. • The `stats.battery` attribute doesn't exist in the shadow's `desired` section.
`shadow.name.<shadowName>.hasDelta:true`	Queries for things that have a shadow with the given name and also a delta element.
`shadow.name.<shadowName>.desired.filament:*`	Queries for things that have a shadow with the given name and also a desired filament property.
`shadow.name.<shadowName>.reported.location:*`	Queries for things that have a shadow with the given name and where the `location` attribute exists in the named shadow's reported section.
`connectivity.connected:true`	Queries for all connected devices.
`connectivity.connected:false`	Queries for all disconnected devices.

Query string	Result
`connectivity.connected:true AND connectivity.timestamp : [1557651600000 TO 1557867600000]`	Queries for all connected devices with a connect timestamp >= 1557651600000 and <= 1557867600000. Timestamps are given in milliseconds since epoch.
`connectivity.connected:false AND connectivity.timestamp : [1557651600000 TO 1557867600000]`	Queries for all disconnected devices with a disconnect timestamp >= 1557651600000 and <= 1557867600000. Timestamps are given in milliseconds since epoch.
`connectivity.connected:true AND connectivity.timestamp > 1557651600000`	Queries for all connected devices with a connect timestamp > 1557651600000. Timestamps are given in milliseconds since epoch.
`connectivity.connected:*`	Queries for all devices with connectivity information present.
`connectivity.disconnectReason:*`	Queries for all devices with connectivity disconnectReason present.
`connectivity.disconnectReason:CLIENT_INITIATED_DISCONNECT`	Queries for all devices disconnected due to CLIENT_INITIATED_DISCONNECT.
`deviceDefender.violationCount:[0 TO 100]`	Queries for things with a Device Defender violations count value that falls within the numeric range (0-100, inclusive).
`deviceDefender.<device-SecurityProfile>.disconnectBehavior.inViolation:true`	Queries for things that are in violation for the behavior `disconnectBehavior` as defined in the security profile `device-SecurityProfile`. Note that *inViolation:false* is not a valid query.

Query string	Result
`deviceDefender.<device-SecurityProfile>.disconnectBehavior.lastViolationValue.number>2`	Queries for things that are in violation for the behavior `disconnectBehavior` as defined in the security profile device-SecurityProfile with a last violation event value greater than 2.
`deviceDefender.<device-SecurityProfile>.disconnectBehavior.lastViolationTime>1634227200000`	Queries for things that are in violation for the behavior `disconnectBehavior` as defined in the security profile device-SecurityProfile with a last violation event after a specified epoch time.
`shadow.name.gps-tracker.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km`	Queries for things that are within the radial distance of 15.5 km from the coordinates of 47.6204,-122.3491. This query string applies to when your location data is stored in a named shadow.
`shadow.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km`	Queries for things that are within the radial distance of 15.5 km from the coordinates of 47.6204,-122.3491. This query string applies to when your location data is stored in a classic shadow.

Example thing group queries

Queries are specified in a query string using a query syntax and passed to the `SearchIndex` API. The following table lists some example query strings.

Query string	Result
`abc`	Queries for "abc" in any field.
`thingGroupName:myGroupThingName`	Queries for a thing group with name "myGroupThingName".

Query string	Result
`thingGroupName:my*`	Queries for thing groups with names that begin with "my".
`thingGroupName:ab?`	Queries for thing groups with names that have "ab" plus one additional character (for example: "aba", "abb", "abc", and so on).
`attributes.myAttribute:75`	Queries for thing groups with an attribute named "myAttribute" that has the value 75.
`attributes.myAttribute:[75 TO 80]`	Queries for thing groups with an attribute named "myAttribute" whose value falls within a numeric range (75–80, inclusive).
`attributes.myAttribute:[75 TO 80]`	Queries for thing groups with an attribute named "myAttribute" whose value falls within the numeric range (>75 and <=80).
`attributes.myAttribute: ["abcd" TO "abcf"]`	Queries for thing groups with an attribute named "myAttribute" whose value is within an alphanumeric string range. This query returns thing groups with a "serialNumber" attribute with values "abcd", "abce", or "abcf".
`attributes.myAttribute:i*t`	Queries for thing groups with an attribute named "myAttribute" whose value is 'i', followed by any number of characters, followed by 't'.
`attributes.attr1:abc AND attributes.attr2<5 NOT attributes.attr3>10`	Queries for thing groups that combine terms using Boolean expressions. This query returns thing groups that have an attribute named "attr1" with a value "abc", an attribute named "attr2" that's less than 5, and an attribute named "attr3" that's not greater than 10.
`NOT attributes.myAttribute:cde`	Queries for thing groups where the attribute named "myAttribute" is not "cde".

Query string	Result
`parentGroupNames:( myParentThingGroupName)`	Queries for thing groups whose parent group name matches "myParentThingGroupName".
`parentGroupNames:( myParentThingGroupName OR myRootThingGroupName)`	Queries for thing groups whose parent group name matches "myParentThingGroupName" or "myRootThingGroupName".
`parentGroupNames:( myParentThingGroupNa*)`	Queries for thing groups whose parent group name begins with "myParentThingGroupNa".

Indexing location data

You can use AWS IoT fleet indexing to index your devices' last sent location data and search for devices using geoqueries. This feature resolves device monitoring and management use cases such as location tracking and proximity search. Location indexing works similarly to other fleet indexing features, and with additional configurations to specify in your thing indexing.

Common use cases include: search and aggregate devices located within desired geographic boundaries, get location specific insights using query terms related to device metadata and state from indexed data sources, provide a granular view such as filtering results to a specific geographic area to reduce rendering lags within your fleet monitoring maps and track last reported device location, and identify devices that are outside of the desired boundary limits and generate alarms using fleet metrics. To get started with location indexing and geoqueries, see ???.

Supported data formats

AWS IoT fleet indexing supports the following location data formats:

1. **Well-known text representation of coordinate reference systems**

   A string that follows the Geographic information - Well-known text representation of coordinate reference systems format. An example can be `"POINT(long lat)"`.

2. **A string that represents the coordinates**

   A string with the format of `"latitude, longitude"` or `"longitude, latitude"` . If you use `"longitude, latitude"`, you must also specify `order` in `geoLocations`. An example can be `"41.12,-71.34"`.

3. **An object of lat(latitude), lon(longitude) keys**

   This format is applicable to classic shadow and named shadow. Supported keys: `lat`, `latitude`, `lon`, `long`, `longitude`. An example can be `{"lat": 41.12, "lon": -71.34}`.

4. **An array that represents the coordinates**

   An array with the format `[lat,lon]` or `[lon,lat]`. If you use the format `[lon,lat]`, which is the same as the coordinates in GeoJSON (applicable to classic shadow and named shadow), you must also specify `order` in `geoLocations`.

   An example can be:

```
{
  "location": {
    "coordinates": [
      **Longitude**,
      **Latitude**
    ],
    "type": "Point",
    "properties": {
      "country": "United States",
      "city": "New York",
      "postalCode": "*****",
      "horizontalAccuracy": 20,
      "horizontalConfidenceLevel": 0.67,
      "state": "New York",
      "timestamp": "2023-01-04T20:59:13.024Z"
    }
  }
}
```

How to index location data

The following steps show how to update indexing configuration for your location data and use geoqueries to search for devices.

1. **Know where your location data is stored**

   Fleet indexing currently supports indexing location data stored in classic shadows or named shadows.

2. **Use supported location data formats**

   Make sure your location data format follows one of the Supported data formats.

3. **Update indexing configuration**

   At a minimum need, enable thing (registry) indexing configuration. You must also enable indexing on classic shadow or named shadow that contain your location data. When updating your thing indexing, you should include your location data in the indexing configuration.

4. **Create and run geoqueries**

   Depending on your use cases, create geoqueries and run them to search for devices. The geoqeury you compose must follow the Query syntax. You can find some examples in ???.

Update thing indexing configuration

To index location data, you must update indexing configuration and include your location data. Depending on where your location data is stored, follow the steps to update your indexing configuration:

Location data stored in classic shadows

If your location data is stored in a classic shadow, you must set `thingIndexingMode` to be `REGISTRY_AND_SHADOW` and specify your location data in the `geoLocations` fields (`name` and `order`) in `filter`.

In the following thing indexing configuration example, you specify the location data path `shadow.reported.coordinates` as `name` and `LonLat` as `order`.

```
{
 "thingIndexingMode": "REGISTRY_AND_SHADOW",
 "filter": {
  "geoLocations": [
   {
    "name": "shadow.reported.coordinates",
    "order": "LonLat"
```

```
      }
    ]
  }
}
```

- `thingIndexingMode`

  The indexing mode controls if registry or shadow is indexed. When `thingIndexingMode` is set to be `OFF`, thing indexing is disabled.

  To index location data stored in a classic shadow, you must set `thingIndexingMode` to be `REGISTRY_AND_SHADOW`. For more information, see ???.
- `filter`

  Indexing filter provides additional selections for named shadows and geolocation data. For more information, see ???.
- `geoLocations`

  The list of geolocation targets that you select to index. The default maximum number of geolocation targets for indexing is 1. To increase the limit, see AWS IoT Device Management Quotas.
- `name`

  The name of the geolocation target field. An example value of `name` can be the location data path of your shadow: `shadow.reported.coordinates`.
- `order`

  The order of the geolocation target field. Valid values: `LatLon` and `LonLat`. `LatLon` means latitude and longitude. `LonLat` means longitude and latitude. This field is optional. The default value is `LatLon`.

Location data stored in named shadows

If your location data is stored in a named shadow, set `namedShadowIndexingMode` to be `ON`, add your named shadow name(s) to the `namedShadowNames` field in `filter`, and specify your location data path in the `geoLocations` field in `filter`.

In the following thing indexing configuration example, you specify the location data path `shadow.name.namedShadow1.reported.coordinates` as `name` and `LonLat` as `order`.

```
{
 "thingIndexingMode": "REGISTRY",
 "namedShadowIndexingMode": "ON",
 "filter": {
  "namedShadowNames": [
   "namedShadow1"
  ],
  "geoLocations": [
   {
    "name": "shadow.name.namedShadow1.reported.coordinates",
    "order": "LonLat"
   }
  ]
 }
}
```

- `thingIndexingMode`

  The indexing mode controls if registry or shadow is indexed. When `thingIndexingMode` is set to be `OFF`, thing indexing is disabled.

  To index location data stored in a named shadow, you must set `thingIndexingMode` to be `REGISTRY` (or `REGISTRY_AND_SHADOW`). For more information, see ???.

- `filter`

  Indexing filter provides additional selections for named shadows and geolocation data. For more information, see ???.

- `geoLocations`

  The list of geolocation targets that you select to index. The default maximum number of geolocation targets for indexing is 1. To increase the limit, see AWS IoT Device Management Quotas.

- `name`

  The name of the geolocation target field. An example value of `name` can be the location data path of your shadow: `shadow.name.namedShadow1.reported.coordinates`.

- `order`

The order of the geolocation target field. Valid values: `LatLon` and `LonLat`. `LatLon` means latitude and longitude. `LonLat` means longitude and latitude. This field is optional. The default value is `LatLon`.

Example geoqueries

After you complete the indexing configuration for your location data, run geoqueries to search for devices. You can also combine your geoqueries with other query strings. For more information, see ??? and ???.

Example query 1

This example assumes the location data is stored in a named shadow `gps-tracker`. The output of this command is the list of devices that are within the radial distance of 15.5 km from the center point with coordinates (47.6204,-122.3491).

```
aws iot search-index --query-string \
"shadow.name.gps-tracker.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km"
```

Example query 2

This example assumes the location data is stored in a classic shadow. The output of this command is the list of devices that are within the radial distance of 15.5 km from the center point with coordinates (47.6204,-122.3491).

```
aws iot search-index --query-string \
"shadow.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km"
```

Example query 3

This example assumes the location data is stored in a classic shadow. The output of this command is the list of devices that are not connected and outside the radial distance of 15.5 km from the center point with coordinates (47.6204,-122.3491).

```
aws iot search-index --query-string \
"connectivity.connected:false AND (NOT
 shadow.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km)"
```

Getting started tutorial

This tutorial demonstrates how to use fleet indexing to index your location data. For simplicity, you create a thing to represent your device and store the location data in a named shadow, update thing indexing configuration for location indexing, and run example geoqueries to search for devices within a radial boundary.

This tutorial takes about 15 minutes to complete.

In this topic:

Prerequisites

- Install the latest version of AWS CLI.
- Familiarize yourself with Location indexing and geoqueries, Manage thing indexing, and Query syntax.

Create thing and shadow

You create a thing to represent your device, and a named shadow to store its location data (coordinates 47.61564, -122.33584).

1. Run the following command to create your thing that represents your bike named Bike-1. For more information about how to create a thing using AWS CLI, see create-thing from *AWS CLI Reference*.

```
aws iot create-thing --thing-name "Bike-1" \
--attribute-payload '{"attributes": {"model":"OEM-2302-12", "battery":"35",
 "acqDate":"06/09/23"}}'
```

   The output of this command can look like the following:

```
{
```

```
    "thingName": "Bike-1",
    "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/Bike-1",
    "thingId": "df9cf01d-b0c8-48fe-a2e2-e16cff6b23df"
}
```

2. Run the following command to create a named shadow to store Bike-1's location data (coordinates 47.61564, -122.33584). For more information about how to create a named shadow using AWS CLI, see update-thing-shadow from *AWS CLI Reference*.

```
aws iot-data update-thing-shadow \
--thing-name Bike-1 \
--shadow-name Bike1-shadow \
--cli-binary-format raw-in-base64-out \
--payload '{"state":{"reported":{"coordinates":{"lat": 47.6153, "lon": -122.3333}}}}'
 \
"output.txt" \
```

   This command doesn't produce any output. To view the named shadow you created, you can run the list-named-shadows-for-thing CLI command.

```
aws iot-data list-named-shadows-for-thing --thing-name Bike-1
```

   The output of this command can look like the following:

```
{
    "results": [
        "Bike1-shadow"
    ],
    "timestamp": 1699574309
}
```

Update thing indexing configuration

To index your location data, you must update your thing indexing configuration to include the location data. Because your location data is stored in a named shadow in this tutorial, set `thingIndexingMode` to be `REGISTRY` (at a minimum requirement), set `namedShadowIndexingMode` to be `ON`, and add your location data to the configuration. In this example, you must add the name of your named shadow and the shadow's location data path to `filter`.

1. Run the command to update your indexing configuration for location indexing.

```
aws iot update-indexing-configuration --cli-input-json '{
"thingIndexingConfiguration": { "thingIndexingMode": "REGISTRY",
"thingConnectivityIndexingMode": "OFF",
"deviceDefenderIndexingMode": "OFF",
"namedShadowIndexingMode": "ON",
"filter": {
    "namedShadowNames": ["Bike1-shadow"],
    "geoLocations":[{
        "name":"shadow.name.Bike1-shadow.reported.coordinates"
    }]
},
"customFields": [
{ "name":"attributes.battery",
"type":"Number"}] } }'
```

   The command doesn't produce any output. You may need to wait for a moment until the update is complete. To check the status, run the [describe-index](https://docs.aws.amazon.com/cli/latest/reference/iot/describe-index.html) CLI command. If you see `indexStatus` shows: `ACTIVE`, your thing indexing update is complete.

2. Run the command to verify your indexing configuration. This step is optional.

```
aws iot get-indexing-configuration
```

   The output can look like the following:

```
{
    "thingIndexingConfiguration": {
        "thingIndexingMode": "REGISTRY",
        "thingConnectivityIndexingMode": "OFF",
        "deviceDefenderIndexingMode": "OFF",
        "namedShadowIndexingMode": "ON",
        "managedFields": [
            {
                "name": "shadow.name.*.hasDelta",
                "type": "Boolean"
            },
            {
                "name": "registry.version",
                "type": "Number"
            },
```

```
            {
                "name": "registry.thingTypeName",
                "type": "String"
            },
            {
                "name": "registry.thingGroupNames",
                "type": "String"
            },
            {
                "name": "shadow.name.*.version",
                "type": "Number"
            },
            {
                "name": "thingName",
                "type": "String"
            },
            {
                "name": "thingId",
                "type": "String"
            }
        ],
        "customFields": [
            {
                "name": "attributes.battery",
                "type": "Number"
            }
        ],
        "filter": {
            "namedShadowNames": [
                "Bike1-shadow"
            ],
            "geoLocations": [
                {
                    "name": "shadow.name.Bike1-shadow.reported.coordinates",
                    "order": "LatLon"
                }
            ]
        }
    },
    "thingGroupIndexingConfiguration": {
        "thingGroupIndexingMode": "OFF"
    }
}
```

Run geoquery

Now you have updated your thing indexing configuration to include the location data. Try to create some geoqueries and run them to see if you can get your desired search results. A geoquery must follow the Query syntax. You can find some useful example geoqueries in ???.

In the following example command, you use the geoquery `shadow.name.Bike1-shadow.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km` to search for devices that are within the radial distance of 15.5 km from the center point with coordinates (47.6204,-122.3491).

```
aws iot search-index --query-string "shadow.name.Bike1-
shadow.reported.coordinates:geo_distance,47.6204,-122.3491,15.5km"
```

Because you have a device located at the coordinates "lat": 47.6153, "lon": -122.3333, which falls within the distance of 15.5 km of the center point, you should be able to see this device (Bike-1) in the output. The output can look like the following:

```
{
    "things": [
        {
            "thingName": "Bike-1",
            "thingId": "df9cf01d-b0c8-48fe-a2e2-e16cff6b23df",
            "attributes": {
                "acqDate": "06/09/23",
                "battery": "35",
                "model": "OEM-2302-12"
            },
            "shadow": "{\"reported\":{\"coordinates\":{\"lat\":47.6153,\"lon
\":-122.3333}},\"metadata\":{\"reported\":{\"coordinates\":{\"lat\":{\"timestamp
\":1699572906},\"lon\":{\"timestamp\":1699572906}}}},\"hasDelta\":false,\"version\":1}"
        }
    ]
}
```

For more information, see ???.

Fleet metrics

Fleet metrics is a feature of fleet indexing, a managed service that allows you to index, search, and aggregate your devices' data in AWS IoT. You can use fleet metrics, to monitor your fleet devices' aggregate state in CloudWatch over time, including reviewing your fleet devices' disconnection rate or average battery level changes of a specified period.

Using fleet metrics, you can build aggregation queries whose results are continually emitted to CloudWatch as metrics for analyzing trends and creating alarms. For your monitoring tasks, you can specify the aggregation queries of different aggregation types (**Statistics**, **Cardinality**, and **Percentile**). You can save all of your aggregation queries to create fleet metrics for reuse in the future.

Getting started tutorial

In this tutorial, you create a fleet metric to monitor your sensors' temperatures to detect potential anomalies. When creating the fleet metric, you define an aggregation query that detects the number of sensors with temperatures exceeding 80 degrees Fahrenheit. You specify the query to run every 60 seconds and the query results are emitted to CloudWatch, where you can view the number of sensors that have potential high-temperature risks, and set alarms. To complete this tutorial, you'll use AWS CLI.

In this tutorial, you'll learn how to:

- Set up
- Create fleet metrics
- View metrics in CloudWatch
- Clean up resources

This tutorial takes about 15 minutes to complete.

Prerequisites

- Install the latest version of AWS CLI
- Familiarize yourself with Querying for aggregate data
- Familiarize yourself with Using Amazon CloudWatch metrics

Set up

To use fleet metrics, enable fleet indexing. To enable fleet indexing for your things or thing groups with specified data sources and associated configurations, follow the instructions in Managing thing indexing and Managing thing group indexing.

To set up

1. Run the following command to enable fleet indexing and specify the data sources to search from.

```
aws iot update-indexing-configuration \
--thing-indexing-configuration
 "thingIndexingMode=REGISTRY_AND_SHADOW,customFields=[{name=attributes.temperature,type=Num
{name=attributes.rackId,type=String},
{name=attributes.stateNormal,type=Boolean}],thingConnectivityIndexingMode=STATUS" \
```

   The preceding example CLI command enables fleet indexing to support searching registry data, shadow data, and thing connectivity status using the `AWS_Things` index.

   The configuration change can take a few minutes to complete. Verify that your fleet indexing is enabled before you create fleet metrics.

   To check if your fleet indexing has been enabled, run the following CLI command:

```
aws --region us-east-1 iot describe-index --index-name "AWS_Things"
```

   For more information, see Enable thing indexing.

2. Run the following bash script to create ten things and describe them.

```
# Bash script. Type `bash` before running in other shells.

Temperatures=(70 71 72 73 74 75 47 97 98 99)
Racks=(Rack1 Rack1 Rack2 Rack2 Rack3 Rack4 Rack5 Rack6 Rack6 Rack6)
IsNormal=(true true true true true true false false false false)

for ((i=0; i < 10; i++))
do
  thing=$(aws iot create-thing --thing-name "TempSensor$i" --attribute-
payload attributes="{temperature=${Temperatures[@]:$i:1},rackId=${Racks[@]:
$i:1},stateNormal=${IsNormal[@]:$i:1}}")
  aws iot describe-thing --thing-name "TempSensor$i"
done
```

   This script creates ten things to represent ten sensors. Each thing has attributes of `temperature`, `rackId`, and `stateNormal` as described in the following table:

Attribute	Data type	Description
`temperature`	Number	Temperature value in Fahrenheit
`rackId`	String	ID of the server rack that contains sensors
`stateNormal`	Boolean	Whether the sensor's temperature value is normal or not

The output of this script contains ten JSON files. One of the JSON file looks like the following:

```
{
    "version": 1,
    "thingName": "TempSensor0",
    "defaultClientId": "TempSensor0",
    "attributes": {
        "rackId": "Rack1",
        "stateNormal": "true",
        "temperature": "70"
    },
    "thingArn": "arn:aws:iot:region:account:thing/TempSensor0",
    "thingId": "example-thing-id"
}
```

For more information, see Create a thing.

Create fleet metrics

To create a fleet metric

1. Run the following command to create a fleet metric named *high_temp_FM*. You create the fleet metric to monitor the number of sensors with temperatures exceeding 80 degrees Fahrenheit in CloudWatch.

```
aws iot create-fleet-metric --metric-name "high_temp_FM" --query-string
 "thingName:TempSensor* AND attributes.temperature >80" --period 60 --aggregation-
field "attributes.temperature" --aggregation-type name=Statistics,values=count
```

--metric-name

Data type: string. The `--metric-name` parameter specifies a fleet metric name. In this example, you're creating a fleet metric named *high_temp_FM*.

--query-string

Data type: string. The `--query-string` parameter specifies the query string. In this example, the query string means to query all the things with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit. For more information, see Query syntax.

--period

Data type: integer. The `--period` parameter specifies the time to retrieve the aggregated data in seconds. In this example, you specify that the fleet metric you're creating retrieves the aggregated data every 60 seconds.

--aggregation-field

Data type: string. The `--aggregation-field` parameter specifies the attribute to evaluate. In this example, the temperature attribute is to be evaluated.

--aggregation-type

The `--aggregation-type` parameter specifies the statistical summary to display in the fleet metric. For your monitoring tasks, you can customize the aggregation query properties for the different aggregation types (**Statistics**, **Cardinality**, and **Percentile**). In this example, you specify **count** for the aggregation type and **Statistics** to return the count of devices that have attributes that match the query, in other words, to return the count of the devices with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit. For more information, see Querying for aggregate data.

The output of this command looks like the following:

```
{
```

```
    "metricArn": "arn:aws:iot:region:111122223333:fleetmetric/high_temp_FM",
    "metricName": "high_temp_FM"
}
```

> **Note**
>
> It can take a moment for the data points to display in CloudWatch.

To learn more about how to create a fleet metric, read [Managing fleet metrics](#).

If you can't create a fleet metric, read [Troubleshooting fleet metrics](#).

2. (Optional) Run the following command to describe your fleet metric named *high_temp_FM*:

```
aws iot describe-fleet-metric --metric-name "high_temp_FM"
```

The output of this command looks like the following:

```
{
    "queryVersion": "2017-09-30",
    "lastModifiedDate": 1625249775.834,
    "queryString": "*",
    "period": 60,
    "metricArn": "arn:aws:iot:region:111122223333:fleetmetric/high_temp_FM",
    "aggregationField": "registry.version",
    "version": 1,
    "aggregationType": {
        "values": [
            "count"
        ],
        "name": "Statistics"
    },
    "indexName": "AWS_Things",
    "creationDate": 1625249775.834,
    "metricName": "high_temp_FM"
}
```

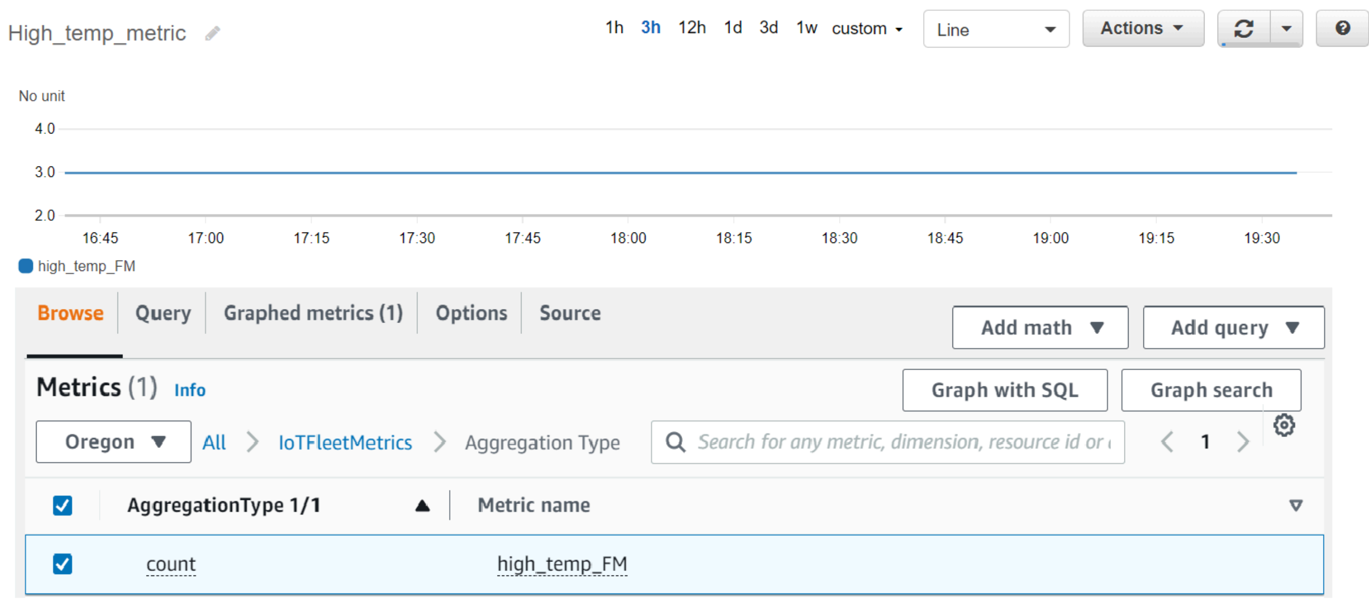
View fleet metrics in CloudWatch

After creating the fleet metric, you can view the metric data in CloudWatch. In this tutorial, you will see the metric that shows the number of sensors with names starting with *TempSensor* and with temperatures higher than 80 degrees Fahrenheit.

To view data points in CloudWatch

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. On the CloudWatch menu on the left panel, choose **Metrics** to expand the submenu and then choose **All metrics**. This opens the page with the upper half to display the graph and the lower half containing four tabbed sections.
3. The first tabbed section **All metrics** lists all the metrics that you can view in groups, choose **IoTFleetMetrics**. This contains all of your fleet metrics.
4. On the **Aggregation type** section of the **All metrics** tab, choose **Aggregation type** to view all the fleet metrics you created.
5. Choose the fleet metric to display graph on the left of the **Aggregation type** section. You will see the value *count* to the left of your **Metric name**, and this is the value of the aggregation type that you specified in the Create fleet metrics section of this tutorial.
6. Choose the second tab named **Graphed metrics** to the right of the **All metrics** tab to view the fleet metric you chose from the previous step.

   You should be able to see a graph that displays the number of sensors with temperatures higher than 80 degrees Fahrenheit like the following:



> **Note**
>
> The **Period** attribute in CloudWatch defaults to 5 minutes. It's the time interval between data points displaying in CloudWatch. You can change the **Period** setting based on your needs.

7. (Optional) You can set a metric alarm.

    1. On the CloudWatch menu on the left panel, choose **Alarms** to expand the submenu and then choose **All alarms**.
    2. On the **Alarms** page, choose **Create alarm** on the upper right corner. Follow the **Create alarm** instructions in console to create an alarm as needed. For more information, see Using Amazon CloudWatch alarms.

To learn more, read Using Amazon CloudWatch metrics.

If you can't see data points in CloudWatch, read Troubleshooting fleet metrics.

Clean up

To delete fleet metrics

You use the **delete-fleet-metric** CLI command to delete fleet metrics.

To delete the fleet metric named *high_temp_FM*, run the following command.

```
aws iot delete-fleet-metric --metric-name "high_temp_FM"
```

To clean up things

You use the **delete-thing** CLI command to delete things.

To delete the ten things that you created, run the following script:

```
# Bash script. Type `bash` before running in other shells.

for ((i=0; i < 10; i++))
do
  thing=$(aws iot delete-thing --thing-name "TempSensor$i")
```

```
done
```

To clean up metrics in CloudWatch

CloudWatch doesn't support metrics deletion. Metrics expire based on their retention schedules. To learn more, Using Amazon CloudWatch metrics.

Managing fleet metrics

This topic shows how to use the AWS IoT console and AWS CLI to manage your fleet metrics.

Topics

Managing fleet metrics (Console)

The following sections show how to use the AWS IoT console to manage your fleet metrics. Make sure you've enabled fleet indexing with associated data sources and configurations before creating fleet metrics.

Enable fleet indexing

If you've already enabled fleet indexing, skip this section.

If you haven't enabled fleet indexing, follow these instructions.

1. Open your AWS IoT console at https://console.aws.amazon.com/iot/.
2. On the AWS IoT menu, choose **Settings**.
3. To view the detailed settings, on the **Settings** page, scroll down to the **Fleet indexing** section.
4. To update your fleet indexing settings, to the right of the **Fleet indexing** section, select **Manage indexing**.
5. On the **Manage fleet indexing** page, update your fleet indexing settings based on your needs.
   - **Configuration**

     To turn on thing indexing, toggle **Thing indexing** on, and then select the data sources you want to index from.

To turn on thing group indexing, toggle **Thing group indexing** on.

- **Custom fields for aggregation - *optional***

  Custom fields are a list of field name and field type pairs.

  To add a custom field pair, choose **Add new field**. Enter a custom field name such as `attributes.temperature`, then select a field type from the **Field type** menu. Note that a custom field name begins with `attributes.` and will be saved as an attribute to run [thing aggregations queries](#).

  To update and save the setting, choose **Update**.

Create a fleet metric

1. Open your AWS IoT console at https://console.aws.amazon.com/iot/.
2. On the AWS IoT menu, choose **Manage**, and then choose **Fleet metrics**.
3. On the **Fleet metrics** page, choose **Create fleet metric** and complete the creation steps.
4. In step 1 **Configure fleet metrics**
   - In **Query** section, enter a query string to specify the things or thing groups you want to perform the aggregate search. The query string consists of an attribute and a value. For **Properties**, choose the attribute you want, or, if it doesn't appear in the list, enter the attribute in the field. Enter the value after `:`. An example query string can be `thingName:TempSensor*`. For each query string you enter, press **enter** in your keyboard. If you enter multiple query strings, specify their relationship by selecting **and**, **or**, **and not**, or **or not** between them.
   - In **Report properties**, choose **Index name**, **Aggregation type**, and **Aggregation field** from their respective lists. Next, select the data you want to aggregate in **Select data**, where you can select multiple data values.
   - Choose **Next**.
5. In step 2 **Specify fleet metric properties**
   - In **Fleet metric name** field, enter a name for the fleet metric you're creating.
   - In **Description - *optional*** field, enter a description for the fleet metric you're creating. This field is optional.
   - In **Hours** and **Minutes** fields, enter the time (how often) you want the fleet metric to emit data to CloudWatch.

- Choose **Next**.

6. In step 3 **Review and create**
   - Review the settings of step 1 and step 2. To edit the settings, choose **Edit**.
   - Choose **Create fleet metric**.

After successful creation, the fleet metric is listed on the **Fleet metric** page.

Update a fleet metric

1. On the **Fleet metric** page, choose the fleet metric that you want to update.
2. On the fleet metric **Details** page, choose **Edit**. This opens the creation steps where you can update your fleet metric in any of the three steps.
3. After you finish updating the fleet metric, choose **Update fleet metric**.

Delete a fleet metric

1. On the **Fleet metric** page, choose the fleet metric that you want to delete.
2. On the next page that shows details of your fleet metric, choose **Delete**.
3. In the dialog box, enter the name of your fleet metric to confirm deletion.
4. Choose **Delete**. This step deletes your fleet metric permanently.

Managing fleet metrics (CLI)

The following sections show how to use the AWS CLI to manage your fleet metrics. Make sure you've enabled fleet indexing with associated data sources and configurations before creating fleet metrics. To enable fleet indexing for your things or thing groups, follow the instructions in Managing thing indexing or Managing thing group indexing.

Create a fleet metric

You can use the create-fleet-metric CLI command to create a fleet metric.

```
aws iot create-fleet-metric --metric-name "YourFleetMetricName" --query-string
 "*" --period 60 --aggregation-field "registry.version" --aggregation-type
 name=Statistics,values=sum
```

The output of this command contains the name and Amazon Resource Name (ARN) of your fleet metric. The output looks like the following:

```
{
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",
    "metricName": "YourFleetMetricName"
}
```

List fleet metrics

You can use the list-fleet-metric CLI command to list all the fleet metrics in your account.

```
aws iot list-fleet-metrics
```

The output of this command contains all your fleet metrics. The output looks like the following:

```
{
    "fleetMetrics": [
        {
            "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/
YourFleetMetric1",
            "metricName": "YourFleetMetric1"
        },
        {
            "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/
YourFleetMetric2",
            "metricName": "YourFleetMetric2"
        }
    ]
}
```

Describe a fleet metric

You can use the describe-fleet-metric CLI command to display more detailed information about a fleet metric.

```
aws iot describe-fleet-metric --metric-name "YourFleetMetricName"
```

The output of command contains the detailed information about the specified fleet metric. The output looks like the following:

```
{
    "queryVersion": "2017-09-30",
    "lastModifiedDate": 1625790642.355,
    "queryString": "*",
    "period": 60,
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",
    "aggregationField": "registry.version",
    "version": 1,
    "aggregationType": {
        "values": [
            "sum"
        ],
        "name": "Statistics"
    },
    "indexName": "AWS_Things",
    "creationDate": 1625790642.355,
    "metricName": "YourFleetMetricName"
}
```

Update a fleet metric

You can use the update-fleet-metric CLI command to update a fleet metric.

```
aws iot update-fleet-metric --metric-name "YourFleetMetricName" --query-string
 "*" --period 120 --aggregation-field "registry.version" --aggregation-type
 name=Statistics,values=sum,count --index-name AWS_Things
```

The update-fleet-metric command doesn't produce any output. You can use the describe-fleet-metric CLI command to see the result.

```
{
    "queryVersion": "2017-09-30",
    "lastModifiedDate": 1625792300.881,
    "queryString": "*",
    "period": 120,
    "metricArn": "arn:aws:iot:us-east-1:111122223333:fleetmetric/YourFleetMetricName",
    "aggregationField": "registry.version",
    "version": 2,
    "aggregationType": {
        "values": [
            "sum",
            "count"
```

```
        ],
        "name": "Statistics"
    },
    "indexName": "AWS_Things",
    "creationDate": 1625792300.881,
    "metricName": "YourFleetMetricName"
}
```

Delete a fleet metric

Use the delete-fleet-metric CLI command to delete a fleet metric.

```
aws iot delete-fleet-metric --metric-name "YourFleetMetricName"
```

This command doesn't produce any output if the deletion is successful or if you specify a fleet metric that doesn't exist.

For more information, see Troubleshooting fleet metrics.

Authorize tagging of IoT resources

For better control over fleet metrics that you can create, modify, or use, you can attach tags to the fleet metrics.

To tag fleet metrics that you create by using AWS Management Console or AWS CLI, you must include the `iot:TagResource` action in your IAM policy to grant the user permissions. If your IAM policy doesn't include `iot:TagResource`, any actions to create a fleet metric with a tag will return an `AccessDeniedException` error.

For general information about tagging your resources, see Tagging your AWS IoT resources.

IAM policy example

Refer to the following IAM policy example granting tagging permissions when you create a fleet metric:

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Action": [
```

```
        "iot:TagResource"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iot:*:*:fleetmetric/*"
      ]
    },
    {
      "Action": [
        "iot:CreateFleetMetric"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iot:*:*:index/*",
        "arn:aws:iot:*:*:fleetmetric/*"
      ]
    }
  ]
}
```

For more information, see Actions, resources, and condition keys for AWS IoT.

MQTT-based file delivery

One option you can use to manage files and transfer them to AWS IoT devices in your fleet is MQTT-based file delivery. With this feature in the AWS Cloud you can create a *stream* that contains multiple files, you can update stream data (the file list and descriptions), get the stream data, and more. AWS IoT MQTT-based file delivery can transfer data in small blocks to your IoT devices, using the MQTT protocol with support for request and response messages in JSON or CBOR.

For more information on ways to transfer data to and from IoT devices using AWS IoT, see [Connect devices to AWS IoT](#).

Topics

What is a stream?

In AWS IoT, a *stream* is a publicly addressable resource that is an abstraction for a list of files that can be transferred to an IoT device. A typical stream contains the following information:

- **An Amazon Resource Name (ARN)** that uniquely identifies a stream at a given time. This ARN has the pattern `arn:partition:iot:region:account-ID:stream/stream ID`.
- **A stream ID** that identifies your stream and is used (and usually required) in AWS Command Line Interface (AWS CLI) or SDK commands.
- **A stream description** that provides a description of the stream resource.
- **A stream version** that identifies a particular version of the stream. Because stream data can be modified immediately before devices start the data transfer, the stream version can be used by the devices to enforce a consistency check.
- **A list of files** that can be transferred to devices. For each file in the list, the stream records a file ID, the file size, and the address information of the file, which consists of, for example, the Amazon S3 bucket name, object key, and object version.
- **An AWS Identity and Access Management (IAM) role** that grants AWS IoT MQTT-based file delivery the permission to read stream files stored in data storage.

AWS IoT MQTT-based file delivery provides the following functionality so that devices can transfer data from the AWS Cloud:

- Data transfer using the MQTT protocol.
- Support for JSON or CBOR formats.
- The ability to describe a stream (`DescribeStream` API) to get a stream file list, stream version, and related information.
- The ability to send data in small blocks (`GetStream` API) so that devices with hardware constraints can receive the blocks.
- Support for a dynamic block size per request, to support devices that have different memory capacities.
- Optimization for concurrent streaming requests when multiple devices request data blocks from the same stream file.
- Amazon S3 as data storage for stream files.
- Support for data transfer log publishing from AWS IoT MQTT-based file delivery to CloudWatch.

For MQTT-based file delivery quotas, see AWS IoT Core Service Quotas in the *AWS General Reference*.

Managing a stream in the AWS Cloud

AWS IoT provides AWS SDK and AWS CLI commands that you can use to manage a stream in the AWS Cloud. You can use these commands to do the following:

- Create a stream. CLI / SDK
- Describe a stream to get its information. CLI / SDK
- List streams in your AWS account. CLI / SDK
- Update the file list or stream description in a stream. CLI / SDK
- Delete a stream. CLI / SDK

> **Note**
>
> At this time, streams are not visible in the AWS Management Console. You must use the AWS CLI or AWS SDK to manage a stream in AWS IoT. Also, Embedded C SDK is the only SDK that supports MQTT-based file transfers.

Before you use AWS IoT MQTT-based file delivery from your devices, you must ensure the following conditions are met for your devices as shown in the next sections:

- A policy reflecting the correct permissions required for transmitting data via MQTT.
- Your device can connect to the AWS IoT Device Gateway.
- A policy statement stating you can tag resources. If `CreateStream` is called with tags, then `iot:TagResource` is required.

Before you use AWS IoT MQTT-based file delivery from your devices, you must follow the steps in the next sections to make sure that your devices are properly authorized and can connect to the AWS IoT Device Gateway.

Grant permissions to your devices

You can follow the steps in Create an AWS IoT policy to create a device policy or use an existing device policy. Attach the policy to the certificates associated with your devices and add the following permissions to the device policy.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:Receive",
                "iot:Publish"
            ],
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/
${iot:Connection.Thing.ThingName}/streams/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
            "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/streams/*"
            ]
        }
    ]
}
```

Connect your devices to AWS IoT

Devices that use AWS IoT MQTT-based file delivery are required to connect with AWS IoT. AWS IoT MQTT-based file delivery integrates with AWS IoT in the AWS Cloud, so your devices should directly connect to the endpoint of the AWS IoT Data Plane.

Note

The endpoint of the AWS IoT data plane is specific to the AWS account and Region. You must use the endpoint for the AWS account and the Region in which your devices are registered in AWS IoT.

See Connect to AWS IoT Core for more information.

TagResource Usage

The `CreateStream` API action creates a stream for delivering one or more large files in chunks over MQTT.

A successful `CreateStream` API call requires the following permissions:

- `iot:CreateStream`
- `iot:TagResource` (if `CreateStream` is with tags)

The policy supporting those two permissions is shown below:

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Action": [
            "iot:CreateStream",
            "iot:TagResource"
        ],
        "Effect": "Allow",
        "Resource": "arn:aws:iot:us-east-1:123456789012:stream/streamId"
    }
}
```

The `iot:TagResource` policy statement action is required to ensure a user can't create or update a tag on a resource without the proper permissions. Without the specifc policy statement action of `iot:TagResource`, the `CreateStream` API call will return an `AccessDeniedException` if the request comes with tags.

For more information, refer to the following links:

- CreateStream
- TagResource
- Tag

Using AWS IoT MQTT-based file delivery in devices

To initiate the data transfer process, a device must receive an **initial data set**, which includes a stream ID at minimum. You can use an AWS IoT Jobs to schedule data transfer tasks for your devices by including the initial data set in the job document. When a device receives the initial data set, it should then start the interaction with AWS IoT MQTT-based file delivery. To exchange data with AWS IoT MQTT-based file delivery, a device should:

- Use the MQTT protocol to subscribe to the MQTT-based file delivery topics.
- Send requests and then wait to receive the responses using MQTT messages.

You can optionally include a stream file ID and a stream version in the initial data set. Sending a stream file ID to a device can simplify the programming of the device's firmware/software, because it eliminates the need to make a `DescribeStream` request from the device to get this ID. The device can specify the stream version in a `GetStream` request to enforce a consistency check in case the stream has been updated unexpectedly.

Use DescribeStream to get stream data

AWS IoT MQTT-based file delivery provides the `DescribeStream` API to send stream data to a device. The stream data returned by this API includes the stream ID, stream version, stream description and a list of stream files, each of which has a file ID and the file size in bytes. With this information, a device can select arbitrary files to initiate the data transfer process.

> **Note**
>
> You don't need to use the `DescribeStream` API if your device receives all required stream file IDs in the initial data set.

Follow these steps to make a `DescribeStream` request.

1. Subscribe to the "accepted" topic filter `$aws/things/`*`ThingName`*`/streams/`*`StreamId`*`/description/json`.
2. Subscribe to the "rejected" topic filter `$aws/things/`*`ThingName`*`/streams/`*`StreamId`*`/rejected/json`.
3. Publish a `DescribeStream` request by sending a message to `$aws/things/`*`ThingName`*`/streams/`*`StreamId`*`/describe/json`.
4. If the request was accepted, your device receives a `DescribeStream` response on the "accepted" topic filter.
5. If the request was rejected, your device receives the error response on the "rejected" topic filter.

Note

If you replace `json` with `cbor` in the topics and topic filters shown, your device receives messages in the CBOR format, which is more compact than JSON.

DescribeStream request

A typical `DescribeStream` request in JSON looks like the following example.

```
{
    "c": "ec944cfb-1e3c-49ac-97de-9dc4aaad0039"
}
```

- (Optional) "`c`" is the client token field.

  The client token can't be longer than 64 bytes. A client token that is longer than 64 bytes causes an error response and an `InvalidRequest` error message.

DescribeStream response

A `DescribeStream` response in JSON looks like the following example.

```
{
    "c": "ec944cfb-1e3c-49ac-97de-9dc4aaad0039",
    "s": 1,
    "d": "This is the description of stream ABC.",
    "r": [
        {
            "f": 0,
            "z": 131072
        },
        {
            "f": 1,
            "z": 51200
        }
    ]
}
```

- "c" is the client token field. This is returned if it was given in the `DescribeStream` request. Use the client token to associate the response with its request.
- "s" is the stream version as an integer. You can use this version to perform a consistency check with your `GetStream` requests.
- "r" contains a list of the files in the stream.
  - "f" is the stream file ID as an integer.
  - "z" is the stream file size in number of bytes.
- "d" contains the description of the stream.

Get data blocks from a stream file

You can use the `GetStream` API so that a device can receive stream files in small data blocks, so it can be used by those devices that have constraints on processing large block sizes. To receive an entire data file, a device might need to send or receive multiple requests and responses until all data blocks are received and processed.

GetStream request

Follow these steps to make a `GetStream` request.

1. Subscribe to the "accepted" topic filter `$aws/things/ThingName/streams/StreamId/data/json`.
2. Subscribe to the "rejected" topic filter `$aws/things/ThingName/streams/StreamId/rejected/json`.
3. Publish a `GetStream` request to the topic `$aws/things/ThingName/streams/StreamId/get/json`.
4. If the request was accepted, your device will receive one or more `GetStream` responses on the "accepted" topic filter. Each response message contains basic information and a data payload for a single block.
5. Repeat steps 3 and 4 to receive all data blocks. You must repeat these steps if the amount of data requested is larger than 128 KB. You must program your device to use multiple `GetStream` requests to receive all of the data requested.
6. If the request was rejected, your device will receive the error response on the "rejected" topic filter.

Note

- If you replace "json" with "cbor" in the topics and topic filters shown, your device will receive messages in the CBOR format, which is more compact than JSON.
- AWS IoT MQTT-based file delivery limits the size of a block to 128 KB. If you make a request for a block that is more than 128 KB, the request will fail.
- You can make a request for multiple blocks whose total size is greater than 128 KB (for example, if you make a request for 5 blocks of 32 KB each for a total of 160 KB of data). In this case, the request doesn't fail, but your device must make multiple requests to receive all of the data requested. The service will send additional blocks as your device makes additional requests. We recommend that you continue with a new request only after the previous response has been correctly received and processed.
- Regardless of the total size of data requested, you should program your device to initiate retries when blocks are not received, or not received correctly.

A typical `GetStream` request in JSON looks like the following example.

```
{
    "c": "1bb8aaa1-5c18-4d21-80c2-0b44fee10380",
    "s": 1,
    "f": 0,
    "l": 4096,
    "o": 2,
    "n": 100,
    "b": "..."
}
```

- [optional] "`c`" is the client token field.

  The client token can be no longer than 64 bytes. A client token that is longer than 64 bytes causes an error response and an `InvalidRequest` error message.
- [optional] "`s`" is the stream version field (an integer).

  MQTT-based file delivery applies a consistency check based on this requested version and the latest stream version in the cloud. If the stream version sent from a device in a `GetStream` request doesn't match the latest stream version in the cloud, the service sends an error response and a `VersionMismatch` error message. Typically, a device receives the expected (latest) stream version in the initial data set or in the response to `DescribeStream`.

- "`f`" is the stream file ID (an integer in the range 0 to 255).

  The stream file ID is required when you create or update a stream using the AWS CLI or SDK. If a device requests a stream file with an ID that doesn't exist, the service sends an error response and a `ResourceNotFound` error message.

- "`l`" is the data block size in bytes (an integer in the range 256 to 131,072).

  Refer to Build a bitmap for a GetStream request for instructions on how to use the bitmap fields to specify what portion of the stream file will be returned in the `GetStream` response. If a device specifies a block size that is out of range, the service sends an error response and a `BlockSizeOutOfBounds` error message.

- [optional] "o" is the offset of the block in the stream file (an integer in the range 0 to 98,304).

  Refer to Build a bitmap for a GetStream request for instructions on how to use the bitmap fields to specify what portion of the stream file will be returned in the `GetStream` response. The maximum value of 98,304 is based on a 24 MB stream file size limit and 256 bytes for the minimum block size. The default is 0 if not specified.

- [optional] "n" is the number of blocks requested (an integer in the range 0 to 98,304).

  The "n" field specifies either (1) the number of blocks requested, or (2) when the bitmap field ("b") is used, a limit on the number of blocks that will be returned by the bitmap request. This second use is optional. If not defined, it defaults to 131072/*`DataBlockSize`*.

- [optional] "b" is a bitmap that represents the blocks being requested.

  Using a bitmap, your device can request non-consecutive blocks, which makes handling retries following an error more convenient. Refer to Build a bitmap for a GetStream request for instructions on how to use the bitmap fields to specify which portion of the stream file will be returned in the `GetStream` response. For this field, convert the bitmap to a string representing the bitmap's value in hexadecimal notation. The bitmap must be less than 12,288 bytes.

> **Important**
>
> Either "n" or "b" should be specified. If neither of them is specified, the `GetStream` request might not be valid when the file size is less than 131072 bytes (128 KB).

GetStream response

A `GetStream` response in JSON looks like this example for each data block that is requested.

```
{
    "c": "1bb8aaa1-5c18-4d21-80c2-0b44fee10380",
    "f": 0,
    "l": 4096,
    "i": 2,
    "p": "..."
}
```

- "c" is the client token field. This is returned if it was given in the `GetStream` request. Use the client token to associate the response with its request.
- "f" is the ID of the stream file to which the current data block payload belongs.
- "l" is the size of the data block payload in bytes.
- "i" is the ID of the data block contained in the payload. Data blocks are numbered starting from 0.
- "p" contains the data block payload. This field is a string, which represents the value of the data block in Base64 encoding.

Build a bitmap for a GetStream request

You can use the bitmap field (b) in a `GetStream` request to get non-consecutive blocks from a stream file. This helps devices with limited RAM capacity deal with network delivery issues. A device can request only those blocks that were not received or not received correctly. The bitmap determines which blocks of the stream file will be returned. For each bit, which is set to 1 in the bitmap, a corresponding block of the stream file will be returned.

Here's an example of how to specify a bitmap and its supporting fields in a `GetStream` request. For example, you want to receive a stream file in chunks of 256 bytes (the block size). Think of each block of 256 bytes as having a number that specifies its position in the file, starting from 0. So block 0 is the first block of 256 bytes in the file, block 1 is the second, and so on. You want to request blocks 20, 21, 24 and 43 from the file.

Block offset

Because the first block is number 20, specify the offset (field o) as 20 to save space in the bitmap.

Number of blocks

To ensure that your device doesn't receive more blocks than it can handle with limited memory resources, you can specify the maximum number of blocks that should be returned in each message sent by MQTT-based file delivery. Note that this value is disregarded if the bitmap itself specifies less than this number of blocks, or if it would make the total size of the response messages sent by MQTT-based file delivery greater than the service limit of 128 KB per `GetStream` request.

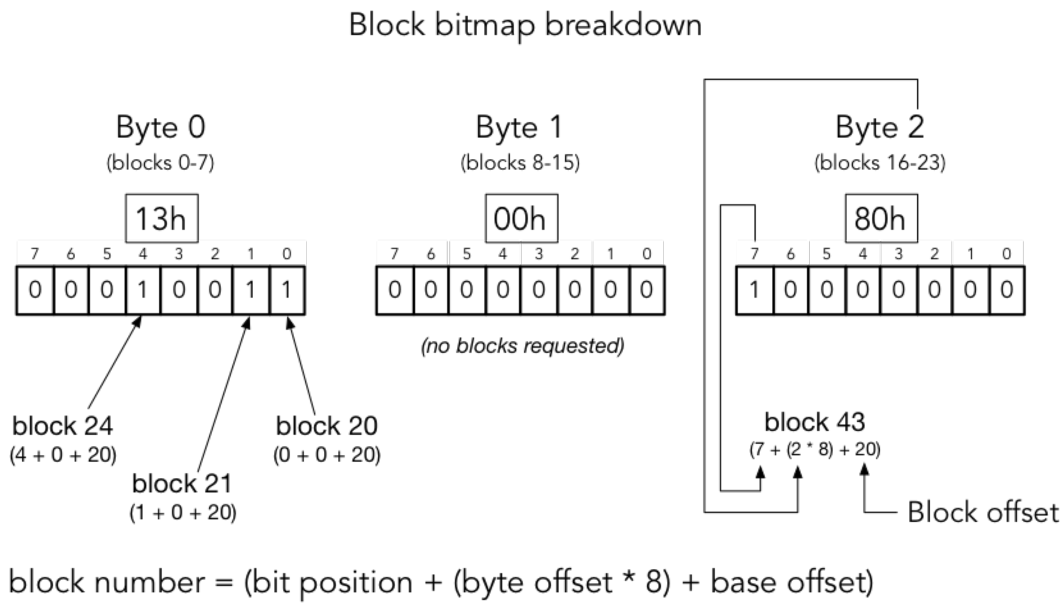
Block bitmap

The bitmap itself is an array of unsigned bytes expressed in hexadecimal notation, and included in the `GetStream` request as a string representation of the number. But to construct this string, let's start by thinking of the bitmap as a long sequence of bits (a binary number). If a bit in this sequence is set to 1, the corresponding block from the stream file will be sent back to the device. For our example, we want to receive blocks 20, 21, 24, and 43, so we must set bits 20, 21, 24, and 43 in our bitmap. We can use the block offset to save space, so after we subtract the offset from each block number, we want to set bits 0, 1, 4, and 23, like the following example.

```
1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Taking one byte (8 bits) at a time, this is conventionally written as: "0b00010011", "0b00000000", and "0b10000000". Bit 0 shows up in our binary representation at the end of the first byte, and bit 23 at the beginning of the last. This can be confusing unless you know the conventions. The first byte contains bits 7-0 (in that order), the second byte contains bits 15-8, the third byte contains bits 23-16, and so on. In hexadecimal notation, this converts to "0x130080".

> **Tip**
>
> You can convert the standard binary to hexadecimal notation. Take four binary digits at a time and convert these to their hexadecimal equivalent. For example, "0001" becomes "1", "0011" becomes "3" and so on.



Putting this all together, the JSON for our `GetStream` request looks like the following.

```
{
    "c" : "1",
    "s" : 1,
    "l" : 256,
    "f" : 1,
    "o" : 20,
    "n" : 32,
    "b" : "130080"
}
```

- "c" is the client token field.
- "s" is the expected stream version.
- "l" is the size of the data block payload in bytes.
- "f" is the ID of the source file index.
- "o" is the block offset.
- "n" is the number of blocks.
- "b" is the missing blockId bitmap starting from the offset. This value must be based64-encoded.

Handling errors from AWS IoT MQTT-based file delivery

An error response that is sent to a device for both `DescribeStream` and `GetStream` APIs contains a client token, an error code and an error message. A typical error response looks like the following example.

```
{
    "o": "BlockSizeOutOfBounds",
    "m": "The block size is out of bounds",
    "c": "1bb8aaa1-5c18-4d21-80c2-0b44fee10380"
}
```

- "o" is the error code that indicates the reason an error occurred. Refer to the error codes later in this section for more details.
- "m" is the error message that contains details of the error.
- "c" is the client token field. This may be returned if it was given in the `DescribeStream` request. You can use the client token to associate the response with its request.

  The client token field is not always included in an error response. When the client token given in the request isn't valid or is malformed, it's not returned in the error response.

> **Note**
>
> For backward compatibility, fields in the error response may be in non-abbreviated form. For example, the error code might be designated by either "code" or "o" fields and the client token field may be designated by either "clientToken" or "c" fields. We recommend that you use the abbreviation form shown above.

InvalidTopic

The MQTT topic of the stream message is invalid.

InvalidJson

The Stream request is not a valid JSON document.

InvalidCbor

The Stream request is not valid CBOR document.

InvalidRequest

The request is generally identified as malformed. For more information, see the error message.

Unauthorized

The request is not authorized to access the stream data files in the storage medium, such as Amazon S3. For more information, see the error message.

BlockSizeOutOfBounds

The block size is out of bounds. Refer to the "**MQTT-based File Delivery**" section in AWS IoT Core Service Quotas.

OffsetOutOfBounds

The offset is out of bounds. Refer to the "**MQTT-based File Delivery**" section in AWS IoT Core Service Quotas.

BlockCountLimitExceeded

The number of request block(s) is out of bounds. Refer to the "**MQTT-based File Delivery**" section in AWS IoT Core Service Quotas.

BlockBitmapLimitExceeded

The size of the request bitmap is out of bounds. Refer to the "**MQTT-based File Delivery**" section in AWS IoT Core Service Quotas.

ResourceNotFound

The requested stream, files, file versions or blocks were not found. Refer to the error message for more details.

VersionMismatch

The stream version in the request doesn't match with the stream version in the MQTT-based file delivery feature. This indicates that the stream data had been modified since the stream version was initially received by the device.

ETagMismatch

The S3 ETag in the stream doesn't match with the ETag of the latest S3 object version.

InternalError

An internal error occurred in MQTT-based file delivery.

An example use case in FreeRTOS OTA

The FreeRTOS OTA (over-the-air) agent uses AWS IoT MQTT-based file delivery to transfer FreeRTOS firmware images to FreeRTOS devices. To send the initial data set to a device, it uses the AWS IoT Job service to schedule an OTA update job to FreeRTOS devices.

For a reference implementation of an MQTT-based file delivery client, see FreeRTOS OTA agent codes in the FreeRTOS documentation.

Device Advisor

Device Advisor is a cloud-based, fully managed test capability for validating IoT devices during device software development. Device Advisor provides pre-built tests that you can use to validate IoT devices for reliable and secure connectivity with AWS IoT Core, before deploying devices to production. Device Advisor's pre-built tests help you validate your device software against best practices for usage of TLS, MQTT, Device Shadow, and IoT Jobs. You can also download signed qualification reports to submit to the AWS Partner Network to get your device qualified for the AWS Partner Device Catalog without the need to send your device in and wait for it to be tested.

> **Note**
>
> Device Advisor is supported in us-east-1, us-west-2, ap-northeast-1, and eu-west-1 regions.
> Device Advisor supports devices and clients that use the MQTT and the MQTT over WebSocket Secure (WSS) protocols to publish and subscribe to messages. All protocols support IPv4 and IPv6.
> Device Advisor supports RSA server certificates.

Any device that has been built to connect to AWS IoT Core can take advantage of Device Advisor. You can access Device Advisor from the AWS IoT console, or by using the AWS CLI or SDK. When you're ready to test your device, register it with AWS IoT Core and configure the device software with the Device Advisor endpoint. Then choose the prebuilt tests, configure them, run the tests on your device, and get the test results along with detailed logs or a qualification report.

Device Advisor is a test endpoint in the AWS cloud. You can test your devices by configuring them to connect to the test endpoint provided by the Device Advisor. After a device is configured to connect to the test endpoint, you can visit the Device Advisor's console or use the AWS SDK to choose the tests you want to run on your devices. Device Advisor then manages the full lifecycle of a test, including the provisioning of resources, scheduling of the test process, managing the state machine, recording the device behavior, logging the results, and providing the final results in form of a test report.

TLS protocols

Transport Layer Security (TLS) protocol is used to encrypt confidential data over insecure networks like the internet. The TLS protocol is the successor of the Secure Sockets Layer (SSL) protocol.

Device Advisor supports the following TLS protocols:

- TLS 1.3 (recommended)
- TLS 1.2

Protocols, port mappings, and authentication

The device communication protocol is used by a device or a client to connect to the message broker by using a device endpoint. The following table lists the protocols that the Device Advisor endpoints support and the authentication methods and ports used.

Protocols, authentication, and port mappings

Protocol	Operations supported	Authentication	Port	ALPN protocol name
MQTT over WebSocket	Publish, Subscribe	Signature Version 4	443	N/A
MQTT	Publish, Subscribe	X.509 client certificate	8883	`x-amzn-mqtt-ca`
MQTT	Publish, Subscribe	X.509 client certificate	443	N/A

This chapter contains the following sections:

- Setting up
- Getting started with Device Advisor in the console
- Device Advisor workflow
- Device Advisor detailed console workflow
- Long duration tests console workflow
- Device Advisor VPC endpoints (AWS PrivateLink)
- Device Advisor test cases

Setting up

Before you use Device Advisor for the first time, complete the following tasks:

Create an IoT thing

First, create an IoT thing and attach a certificate to that thing. For a tutorial on how to create things, see Create a thing object.

Create an IAM role to use as your device role

> **Note**
>
> You can quickly create the device role with the Device Advisor console. To learn how to set up your device role with the Device Advisor console, see Getting started with the Device Advisor in the console.

1. Go to the AWS Identity and Access Management console and log in to the AWS account you use for Device Advisor testing.
2. In the left navigation pane, chose **Policies**.
3. Choose **Create policy**.
4. Under **Create policy**, do the following:
    a. For **Service**, choose **IoT**.
    b. Under **Actions**, do one of the following:
        - (Recommended) Select actions based on the policy attached to the IoT thing or certificate you created in the previous section.
        - Search for the following actions in the **Filter action** box and select them:
            - `Connect`
            - `Publish`
            - `Subscribe`
            - `Receive`
            - `RetainPublish`

c. Under **Resources**, restrict the client, topic, and topic resources. Restricting these resources is a security best practice. To restrict resources, do the following:

   i. Choose **Specify client resource ARN for the Connect action**.

   ii. Choose **Add ARN**, then do either of the following:

   > **Note**
   >
   > The *clientId* is the MQTT client ID that your device uses to interact with Device Advisor.

   - Specify the **Region**, **accountID**, and **clientID** in the visual ARN editor.
   - Manually enter the Amazon Resource Names (ARNs) of the IoT topics you want to run your test cases with.

   iii. Choose **Add**.

   iv. Choose **Specify topic resource ARN for the Receive and one more action**.

   v. Choose **Add ARN**, then do either of the following:

   > **Note**
   >
   > The *topic name* is the MQTT topic that your device publishes messages to.

   - Specify the **Region**, **accountID**, and **Topic name** in the visual ARN editor.
   - Manually enter the ARNs of the IoT topics you want to run your test cases with.

   vi. Choose **Add**.

   vii. Choose **Specify topicFilter resource ARN for the Subscribe action**.

   viii. Choose **Add ARN**, then do either of the following:

   > **Note**
   >
   > The *topic name* is the MQTT topic that your device subscribes to.

   - Specify the **Region**, **accountID**, and **Topic name** in the visual ARN editor.

- Manually enter the ARNs of the IoT topics you want to run your test cases with.

ix. Choose **Add**.

5. Choose **Next: Tags**.
6. Choose **Next: Review**.
7. Under **Review policy**, enter a **Name** for your policy.
8. Choose **Create policy**.
9. On the left navigation pane, Choose **Roles**.
10. Choose **Create Role**.
11. Under **Select trusted entity**, choose **Custom trust policy**.
12. Enter the following trust policy into the **Custom trust policy** box. To protect against the confused deputy problem, add the global condition context keys `aws:SourceArn` and `aws:SourceAccount` to the policy.

> **Important**
>
> Your `aws:SourceArn` must comply with the `format:` `arn:aws:iotdeviceadvisor:`*`region:account-id`*`:*`. Make sure that *`region`* matches your AWS IoT Region and *`account-id`* matches your customer account ID. For more information, see Cross-service confused deputy prevention.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAwsIoTCoreDeviceAdvisor",
            "Effect": "Allow",
            "Principal": {
                "Service": "iotdeviceadvisor.amazonaws.com"
        },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "123456789012"
            },
                "ArnLike": {
```

```
                        "aws:SourceArn":
 "arn:aws:iotdeviceadvisor:*:123456789012:suitedefinition/*"
                    }
               }
               }
          ]
}
```

13. Choose **Next**.
14. Choose the policy you created in Step 4.
15. (Optional) Under **Set permissions boundary**, choose **Use a permissions boundary to control the maximum role permissions**, and then select the policy you created.
16. Choose **Next**.
17. Enter a **Role name** and a **Role description**.
18. Choose **Create role**.

Create a custom-managed policy for an IAM user to use Device Advisor

1. Navigate to the IAM console at https://console.aws.amazon.com/iam/. If prompted, enter your AWS credentials to sign in.
2. In the left navigation pane, choose **Policies**.
3. Choose **Create Policy**, then choose the **JSON** tab.
4. Add the necessary permissions to use Device Advisor. The policy document can be found in the topic Security best practices.
5. Choose **Review Policy**.
6. Enter a **Name** and **Description**.
7. Choose **Create Policy**.

Create an IAM user to use Device Advisor

Note

We recommend that you create an IAM user to use when you run Device Advisor tests. Running Device Advisor tests from an admin user can pose security risks and isn't recommended.

1. Navigate to the IAM console at https://console.aws.amazon.com/iam/ If prompted, enter your AWS credentials to sign in.
2. In the left navigation pane, Choose **Users**.
3. Choose **Add User**.
4. Enter a **User name**.
5. Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

   To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
IAM	(Recommended) Use console credentials as temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Login for AWS local development in the *AWS Command Line Interface User Guide*. • For AWS SDKs, see Login for AWS local development in the *AWS SDKs and Tools Reference Guide*.
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS*

Which user needs programmatic access?	To	By
		Command Line Interface User Guide. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

6. Choose **Next: Permissions**.
7. To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

8. Enter the name of the custom-managed policy that you created in the search box. Then, select the check box for **Policy name**.
9. Choose **Next: Tags**.
10. Choose **Next: Review**.
11. Choose **Create user**.
12. Choose **Close**.

Device Advisor requires access to your AWS resources (things, certificates, and endpoints) on your behalf. Your IAM user must have the necessary permissions. Device Advisor will also publish logs to Amazon CloudWatch if you attach the necessary permissions policy to your IAM user.

Configure your device

Device Advisor uses the server name indication (SNI) TLS extension to apply TLS configurations. Devices must use this extension when they connect and pass a server name that is identical to the Device Advisor test endpoint.

Device Advisor allows the TLS connection when a test is in the `Running` state. It denies the TLS connection before and after each test run. For this reason, we recommend that you use the device connect retry mechanism for a fully automated testing experience with Device Advisor. You can run test suites that include more than one test case, such as TLS connect, MQTT connect, and MQTT publish. If you run multiple test cases, we recommend that your device try to connect to our test endpoint every five seconds. You can then automate running multiple test cases in sequence.

> **Note**
>
> To ready your device software for testing, we recommend that you use an SDK that can connect to AWS IoT Core. You should then update the SDK with the Device Advisor test endpoint provided for your AWS account.

Device Advisor supports two types of endpoints: Account-level and Device-level endpoints. Choose the endpoint that best fits your use case. To simultaneously run multiple test suites for different devices, use a Device-level endpoint.

Run the following command to get the Device-level endpoint:

For MQTT customers using X.509 client certificates:

```
aws iotdeviceadvisor get-endpoint --thing-arn your-thing-arn
```

or

```
aws iotdeviceadvisor get-endpoint --certificate-arn your-certificate-arn
```

For MQTT over WebSocket customers using Signature Version 4:

```
aws iotdeviceadvisor get-endpoint --device-role-arn your-device-role-arn --
authentication-method SignatureVersion4
```

To run one test suite at a time, choose an Account-level endpoint. Run the following command to get the Account-level endpoint:

```
aws iotdeviceadvisor get-endpoint
```

Getting started with Device Advisor in the console

This tutorial helps you get started with AWS IoT Core Device Advisor on the console. Device Advisor offers features such as required tests and signed qualification reports. You can use these tests and reports to qualify and list devices in the AWS Partner Device Catalog as detailed in the AWS IoT Core qualification program.

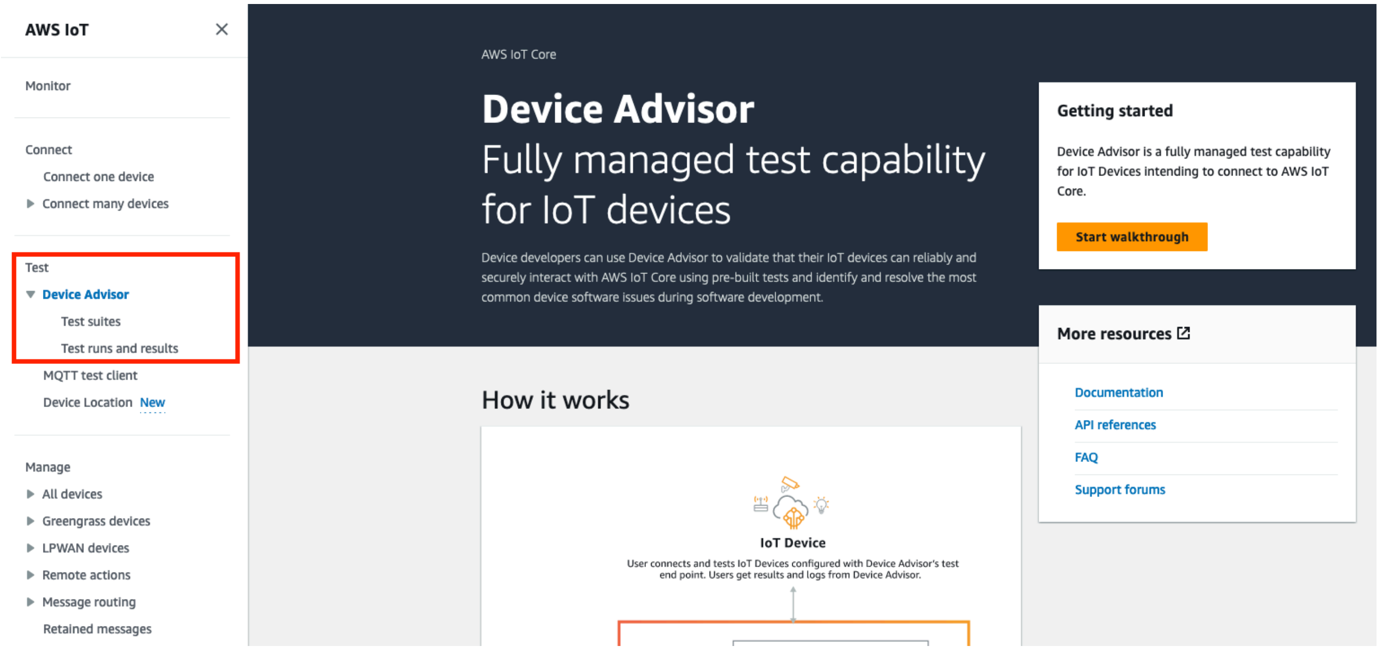
For more information about using Device Advisor, see Device Advisor workflow and Device Advisor detailed console workflow.

To complete this tutorial, follow the steps outlined in Setting up.

> **Note**
>
> Device Advisor is supported in the following AWS Regions:
>
> - US East (N. Virginia)
> - US West (Oregon)
> - Asia Pacific (Tokyo)
> - Europe (Ireland)

Getting started

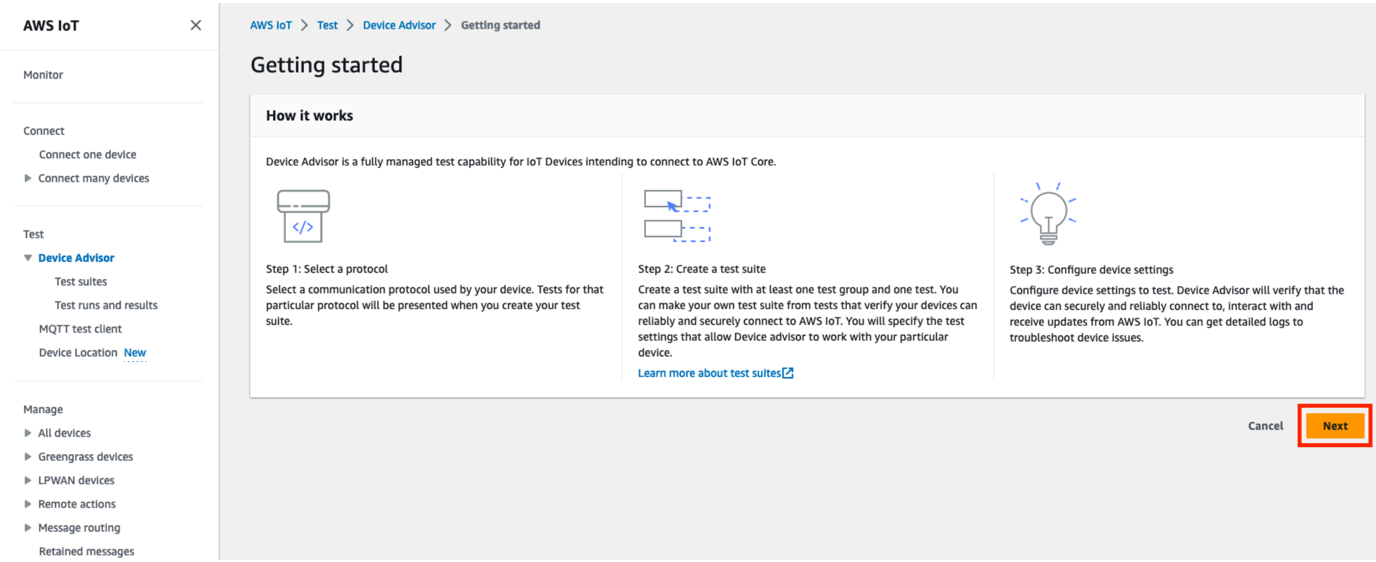
1. In the AWS IoT console's navigation pane under **Test**, choose **Device Advisor**. Then, choose the **Start walkthrough** button on the console.



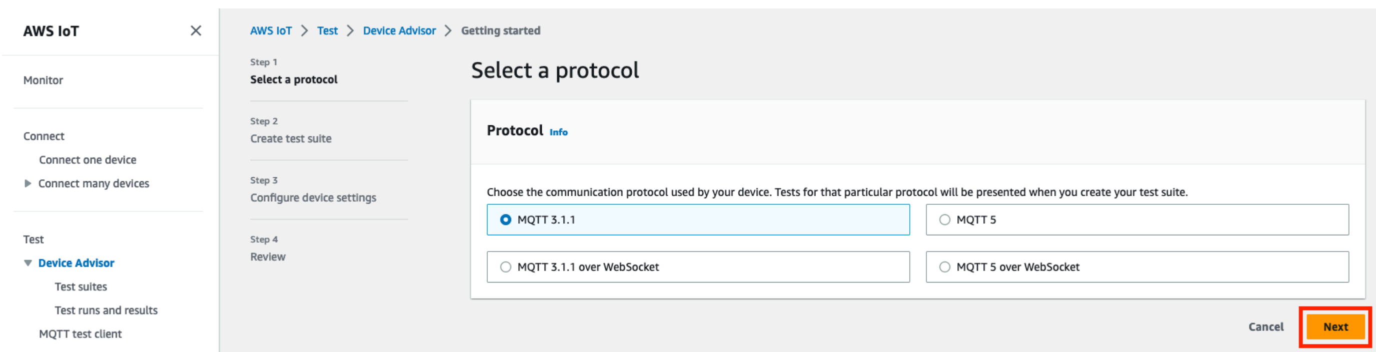
2. The **Getting started with Device Advisor** page provides an overview of the steps required to create a test suite and run tests against your device. You can also find the Device Advisor test

endpoint for your account here. You must configure the firmware or software on the device used for testing to connect to this test endpoint.

To complete this tutorial, first [create a thing and certificate](). After you review the information under **How it works**, choose **Next**.

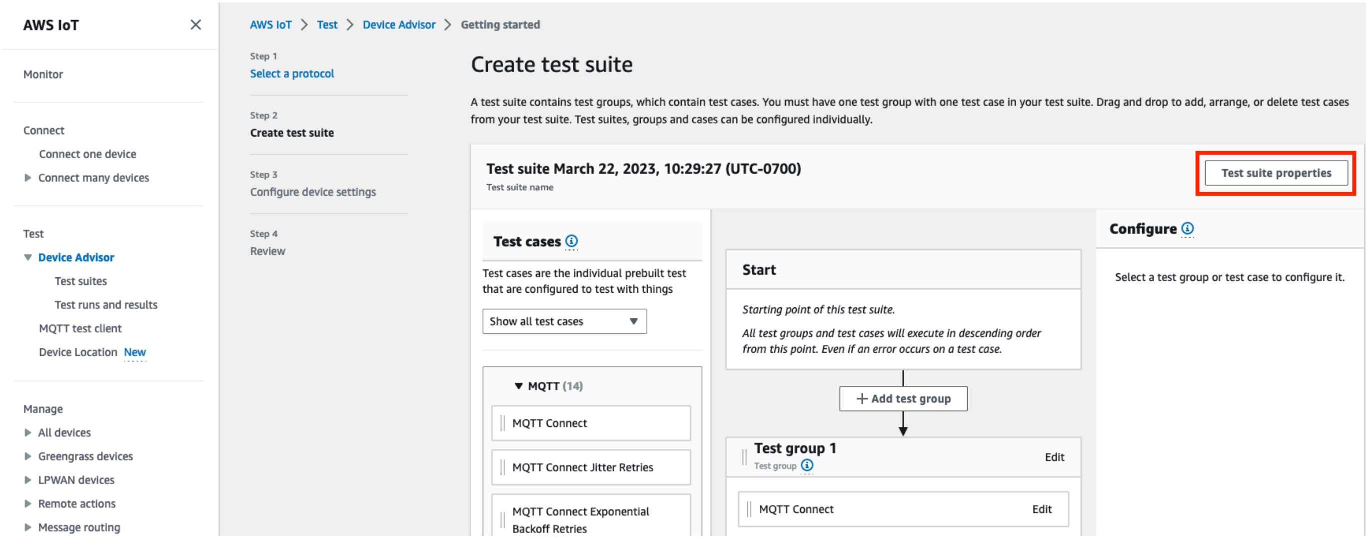


3. In **Step 1: Select a protocol**, select a protocol from the options listed. Then, choose **Next**.



4. In **Step 2**, you create and configure a custom test suite. A custom test suite must have at least one test group, and each test group must have at least one test case. We've added the **MQTT Connect** test case for you to get started.

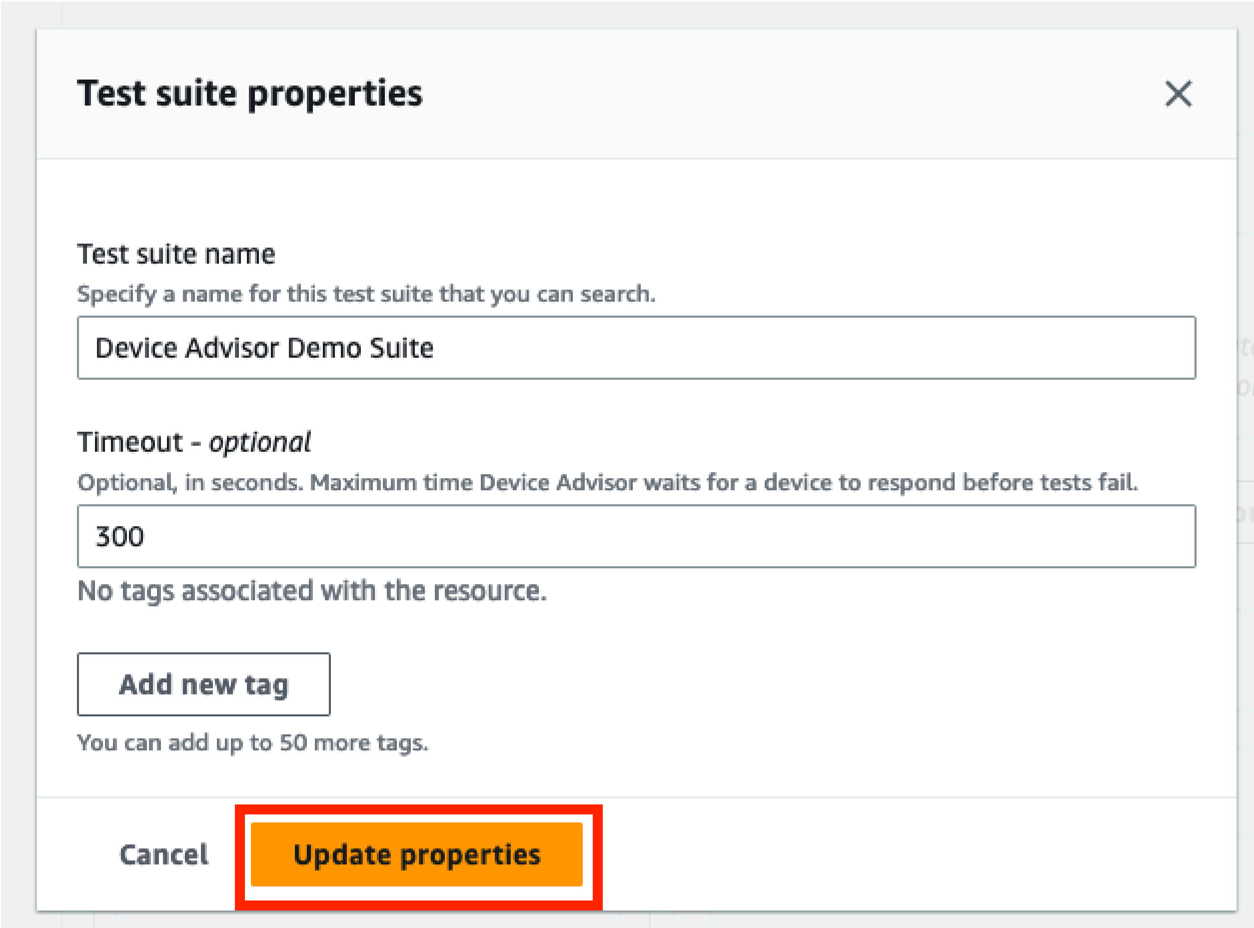
   Choose **Test suite properties**.



Supply the test suite properties when you create your test suite. You can configure the following suite-level properties:

- **Test suite name**: Enter a name for your test suite.
- **Timeout** (optional): The timeout (in seconds) for each test case in the current test suite. If you don't specify a timeout value, the default value is used.
- **Tags** (optional): Add tags to the test suite.

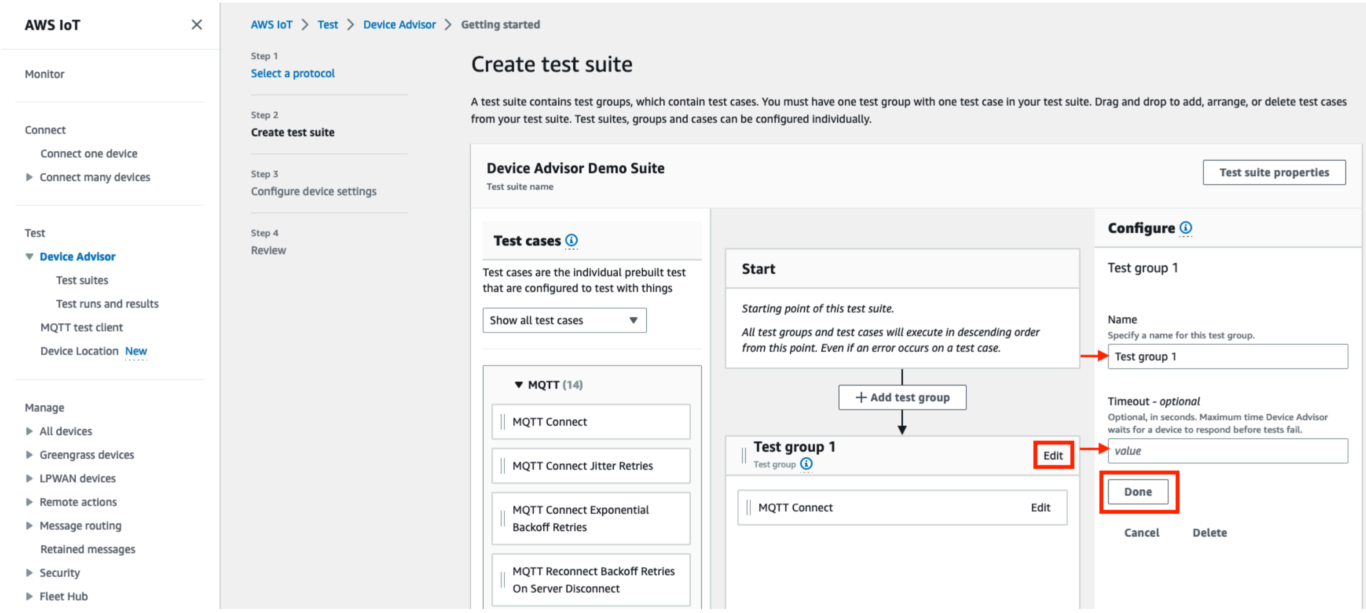
When you’ve finished, choose **Update properties**.



5. (Optional) To update the test suite group configuration, choose the **Edit** button next to the test group name.

   - **Name**: Enter a custom name for the test suite group.
   - **Timeout** (optional): The timeout (in seconds) for each test case in the current test suite. If you don't specify a timeout value, the default value is used.

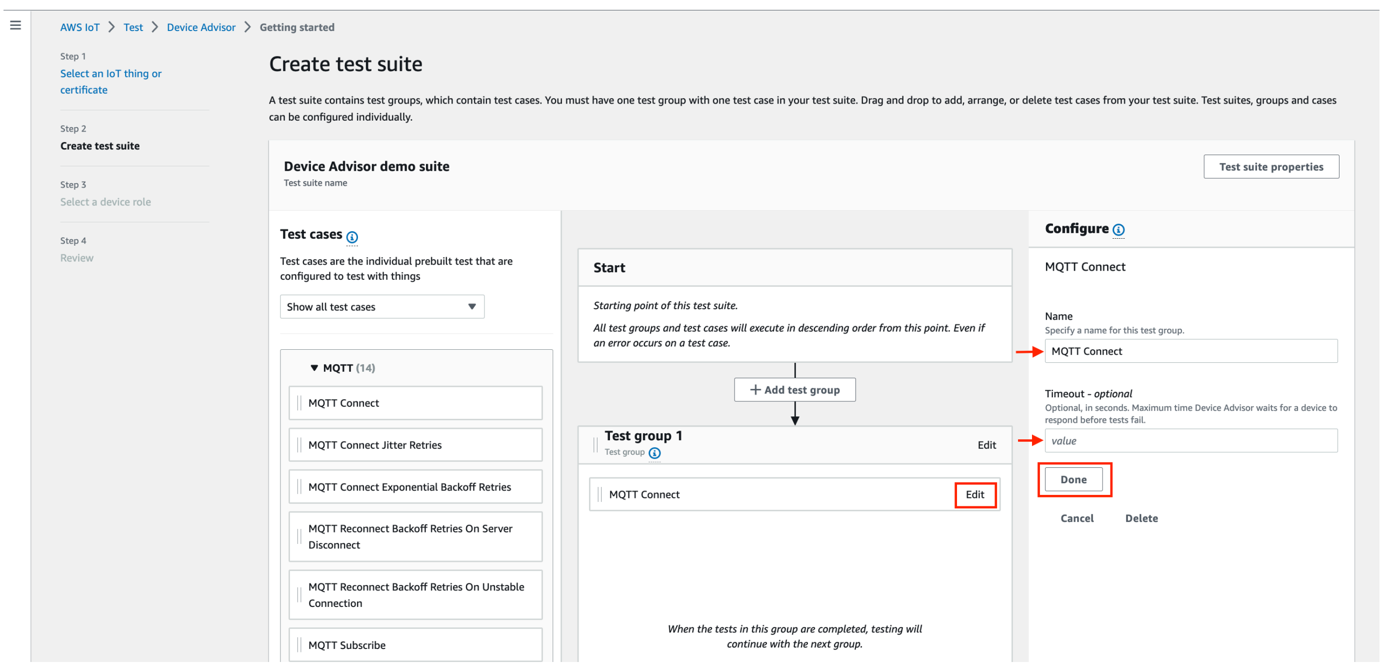
   When finished, choose **Done** to continue.



6. (Optional) To update the test case configuration for a test case, choose the **Edit** button next to the test case name.

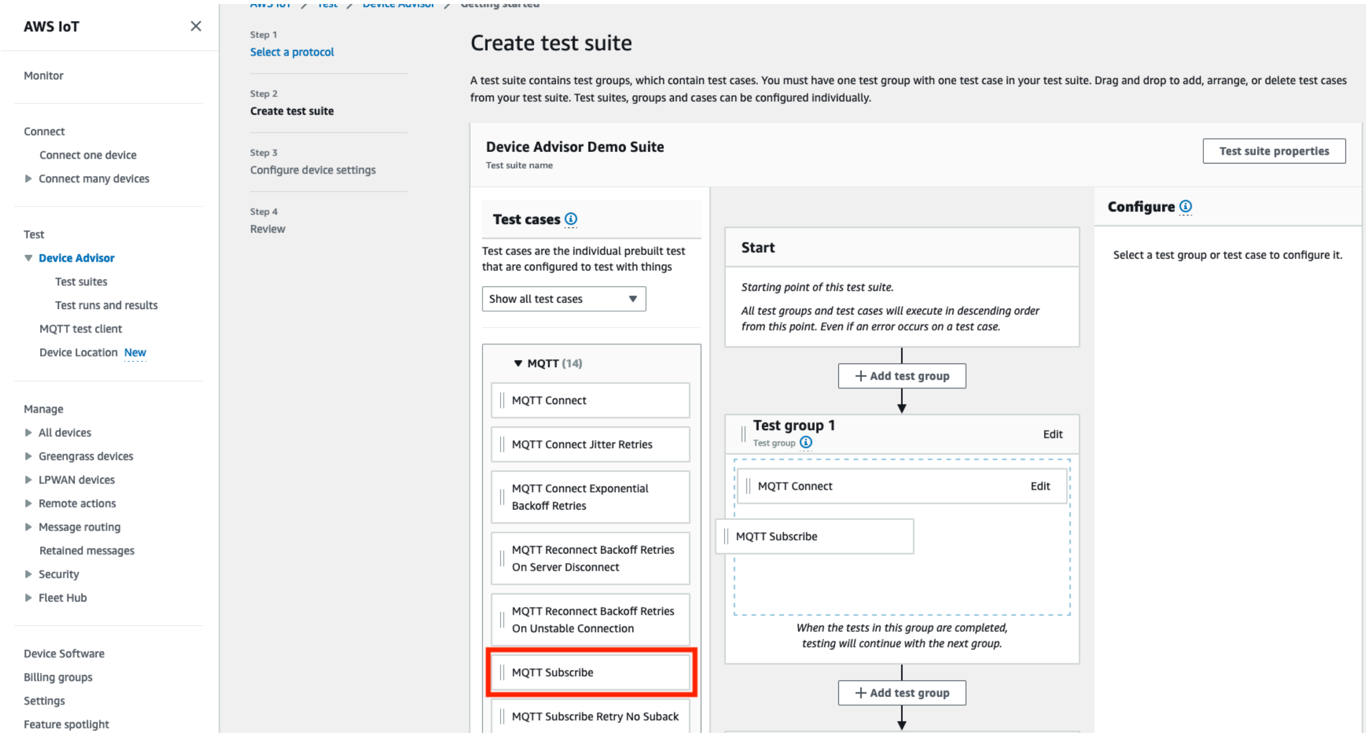
   - **Name**: Enter a custom name for the test suite group.
   - **Timeout** (optional): The timeout (in seconds) for the selected test case. If you don't specify a timeout value, the default value is used.

   When finished, choose **Done** to continue.



7. (Optional) To add more test groups to the test suite, choose **Add test group**, then follow the instructions in Step 5.

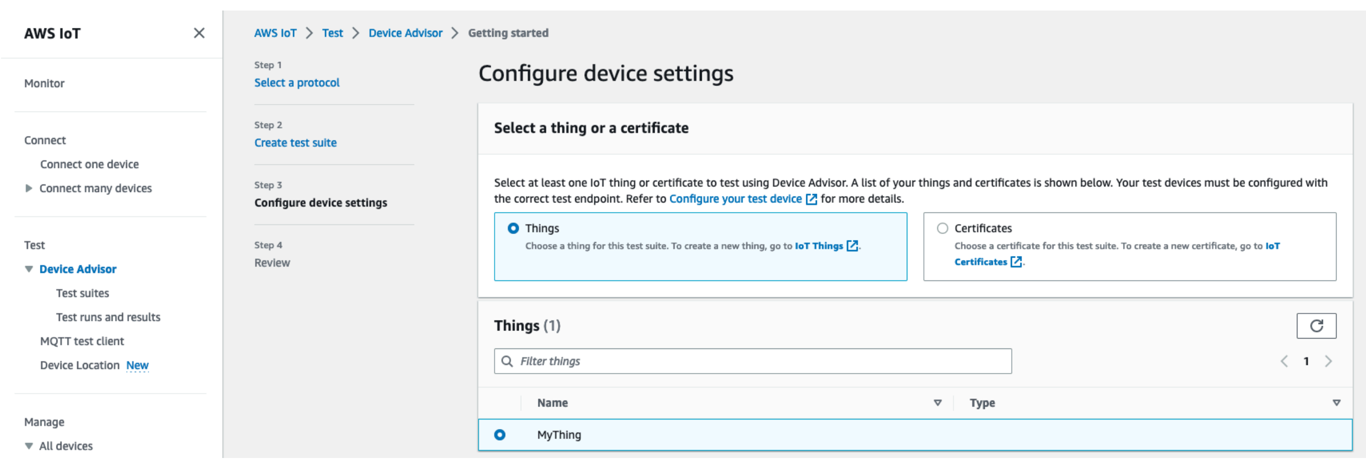
8. (Optional) To add more test cases, drag the test cases in the **Test cases** section into any of your test groups.



9. You can change the order of your test groups and test cases. To make changes, drag the listed test cases up or down the list. Device Advisor runs tests in the order you listed them in.

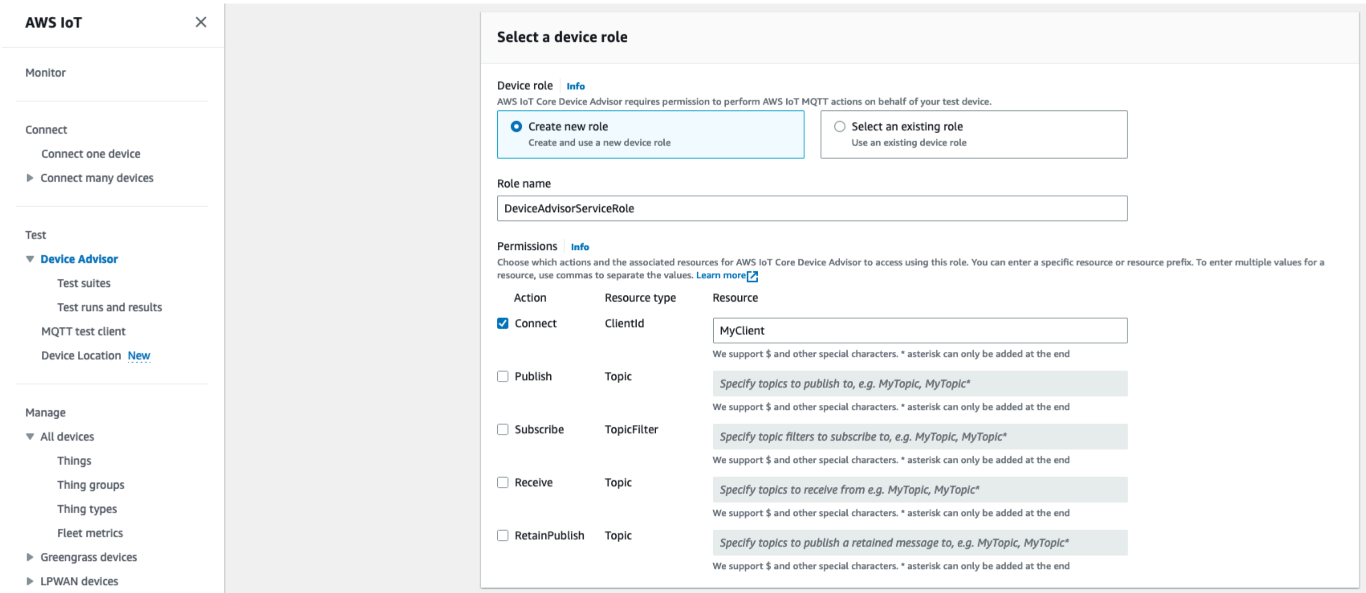
   After you've configured your test suite, choose **Next**.

10. In **Step 3**, select an AWS IoT thing or certificate to test using Device Advisor. If you don't have any existing things or certificates, see [Setting up](#).

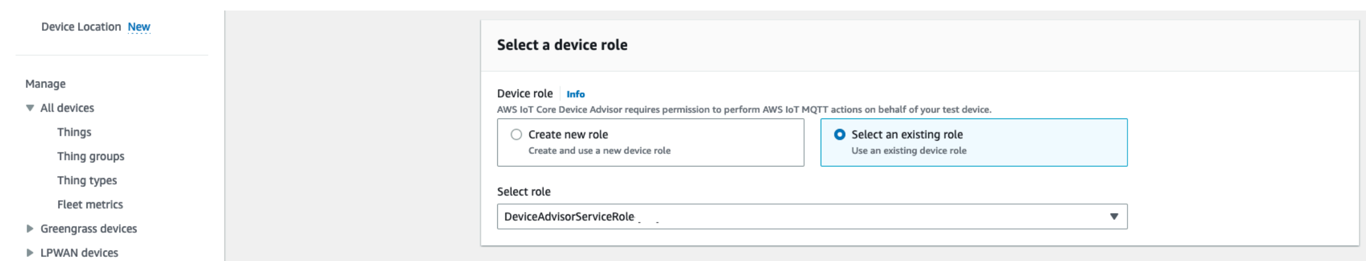


11. You can configure a device role that Device Advisor uses to perform AWS IoT MQTT actions on behalf of your test device. For **MQTT Connect** test case only, the **Connect** action is selected automatically. This is because the device role requires this permission to run the test suite. For other test cases, the corresponding actions are selected.

    Provide the resource values for each of the selected actions. For example, for the **Connect** action, provide the client ID your device uses to connect to the Device Advisor endpoint. You can provide multiple values with comma seperated values, and prefix values with a wildcard (*) character. For example, to provide permission to publish on any topic beginning with `MyTopic`, enter **`MyTopic*`** as the resource value.

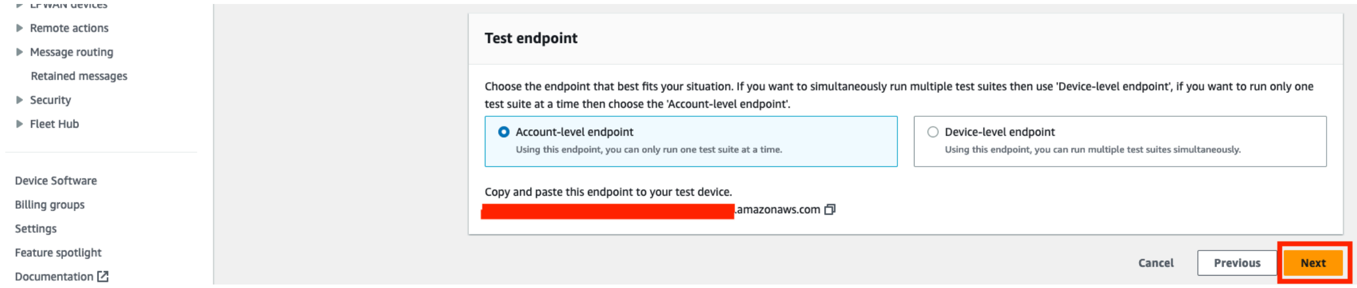


    To use a previously created device role from [Setting up](), choose **Select an existing role**. Then choose your device role under **Select role**.



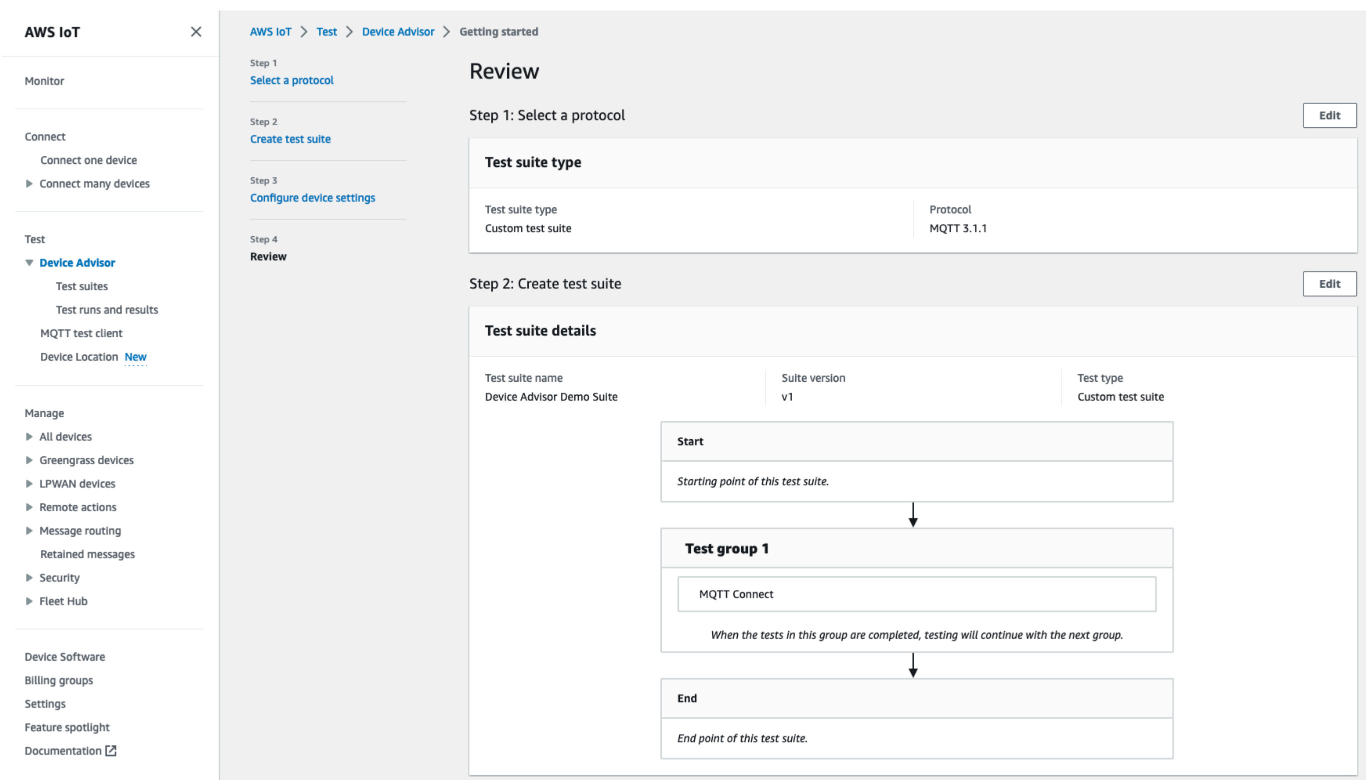
    Configure your device role with one of the two provided options, and then choose **Next**.

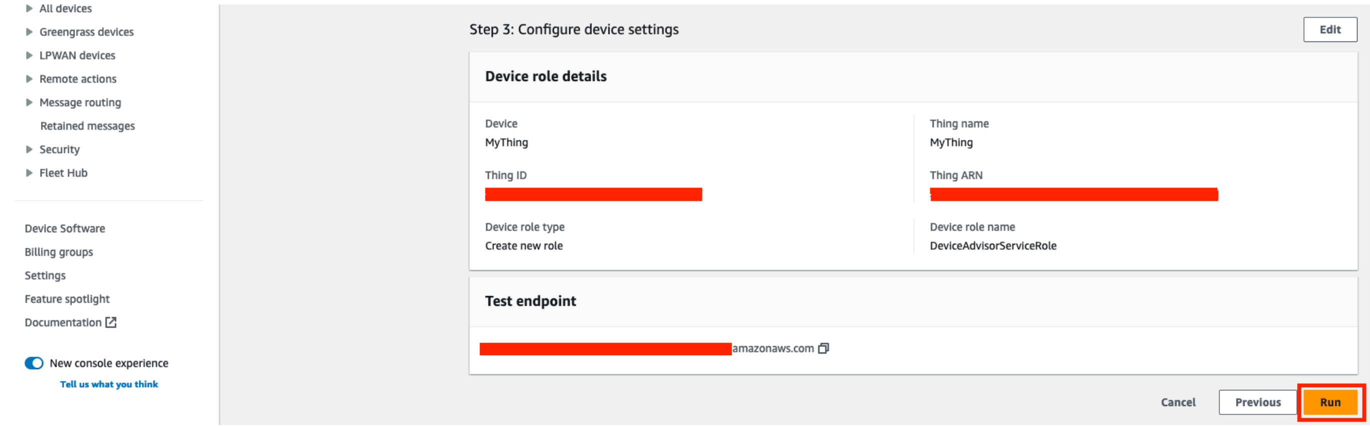
12. In the **Test endpoint** section, select the endpoint that best fits your use case. To run multiple test suites simultaneously with the same AWS account, select **Device-level endpoint**. To run one test suite at a time, select **Account-level endpoint**.



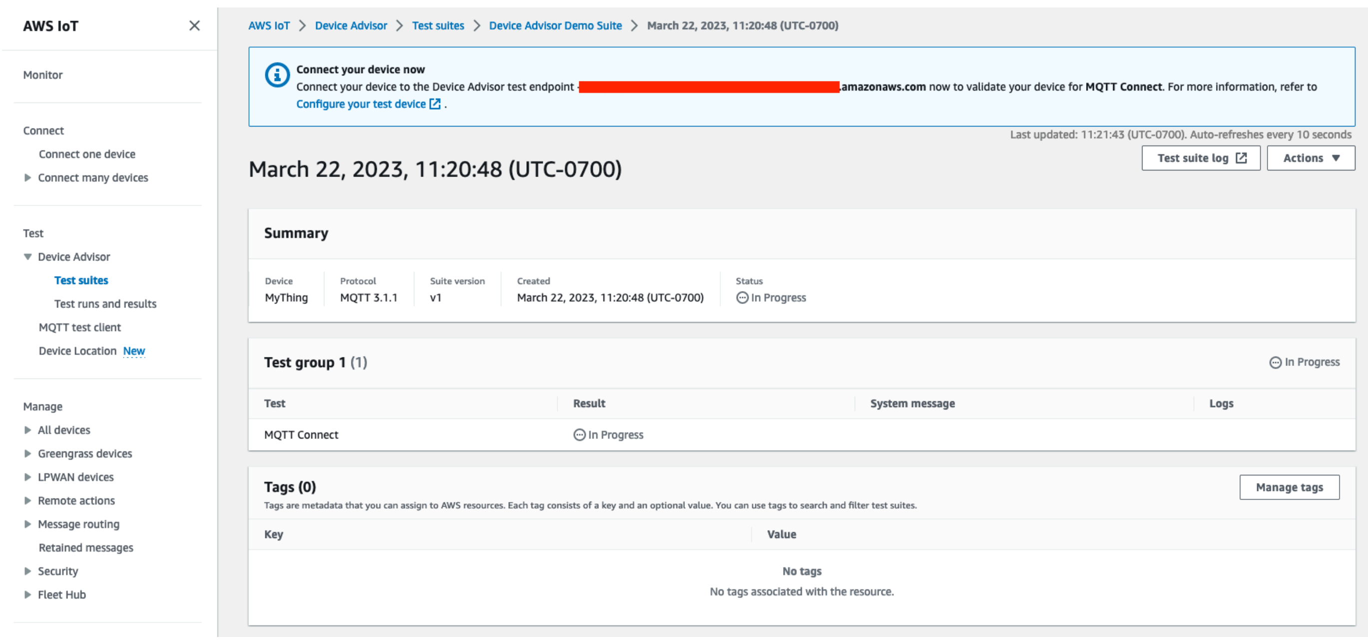
13. **Step 4** shows an overview of the selected test device, test endpoint, test suite, and test device role configured. To make changes to a section, choose the **Edit** button for the section you want to edit. Once you've confirmed your test configuration, choose **Run** to create the test suite and run your tests.

   > **Note**
   >
   > For best results, you can connect your selected test device to the Device Advisor test endpoint before you start the test suite run. We recommend that you have a mechanism built for your device to try connecting to our test endpoint every five seconds for up to one to two minutes.





14. In the navigation pane under **Test**, choose **Device Advisor**, and then choose **Test runs and results**. Select a test suite run to view its run details and logs.



15. To access the Amazon CloudWatch logs for the suite run:

   - Choose **Test suite log** to view the CloudWatch logs for the test suite run.
   - Choose **Test case log** for any test case to view test case-specific CloudWatch logs.

16. Based on your test results, troubleshoot your device until all tests pass.

Device Advisor workflow

This tutorial explains how to create a custom test suite and run tests against the device you want to test in the console. After the tests are complete, you can view the test results and detailed logs.

Prerequisites

Before you begin this tutorial this tutorial, complete the steps outlined in Setting up.

Create a test suite definition

First, install an AWS SDK.

rootGroup syntax

A root group is a JSON string that specifies which test cases to include in your test suite. It also specifies any necessary configurations for those test cases. Use the root group to structure and order your test suite based on your needs. The hierarchy of a test suite is:

```
test suite # test group(s) # test case(s)
```

A test suite must have at least one test group, and each test group must have at least one test case. Device Advisor runs tests in the order in which you define the test groups and test cases.

Each root group follows this basic structure:

```
{
    "configuration": {  // for all tests in the test suite
        "": ""
    }
    "tests": [{
        "name": ""
        "configuration": {  // for all sub-groups in this test group
            "": ""
        },
        "tests": [{
            "name": ""
            "configuration": {  // for all test cases in this test group
                "": ""
            },
            "test": {
                "id": ""
                "version": ""
            }
        }]
    }]
}
```

In the root group, you define the test suite with a `name`, `configuration`, and the `tests` that the group contains. The `tests` group contains the definitions of individual tests. You define each test with a `name`, `configuration`, and a `test` block that defines the test cases for that test. Finally, each test case is defined with an `id` and `version`.

For information on how to use the `"id"` and `"version"` fields for each test case (`test` block), see Device Advisor test cases. That section also contains information on the available `configuration` settings.

The following block is an example of a root group configuration. This configurations specifies the *MQTT Connect Happy Case* and *MQTT Connect Exponential Backoff Retries* test cases, along with descriptions of the configuration fields.

```
{
    "configuration": {},  // Suite-level configuration
    "tests": [            // Group definitions should be provided here
      {
        "name": "My_MQTT_Connect_Group",  // Group definition name
        "configuration": {}               // Group definition-level configuration,
        "tests": [                        // Test case definitions should be provided
 here
        {
            "name": "My_MQTT_Connect_Happy_Case",  // Test case definition name
            "configuration": {
                "EXECUTION_TIMEOUT": 300        // Test case definition-level
 configuration, in seconds
            },
            "test": {
                "id": "MQTT_Connect",              // test case id
                "version": "0.0.0"                 // test case version
            }
        },
        {
            "name": "My_MQTT_Connect_Jitter_Backoff_Retries",  // Test case definition
 name
            "configuration": {
                "EXECUTION_TIMEOUT": 600                 // Test case definition-level
 configuration,  in seconds
            },
            "test": {
                "id": "MQTT_Connect_Jitter_Backoff_Retries",  // test case id
                "version": "0.0.0"                                // test case version
```

```
            }
        }]
    }]
}
```

You must supply the root group configuration when you create your test suite definition. Save the `suiteDefinitionId` that is returned in the response object. You can use this ID to retrieve your test suite definition information and run your test suite.

Here is a Java SDK example:

```
response = iotDeviceAdvisorClient.createSuiteDefinition(
        CreateSuiteDefinitionRequest.builder()
            .suiteDefinitionConfiguration(SuiteDefinitionConfiguration.builder()
                .suiteDefinitionName("your-suite-definition-name")
                .devices(
                    DeviceUnderTest.builder()
                        .thingArn("your-test-device-thing-arn")
                        .certificateArn("your-test-device-certificate-arn")
                        .deviceRoleArn("your-device-role-arn") //if using SigV4 for
 MQTT over WebSocket
                        .build()
                )
                .rootGroup("your-root-group-configuration")
                .devicePermissionRoleArn("your-device-permission-role-arn")
                .protocol("MqttV3_1_1 || MqttV5 || MqttV3_1_1_OverWebSocket ||
 MqttV5_OverWebSocket")
                .build()
            )
            .build()
)
```

Get a test suite definition

After you create your test suite definition, you receive the `suiteDefinitionId` in the response object of the `CreateSuiteDefinition` API operation.

When the operation returns the `suiteDefinitionId`, you may see new `id` fields within each group and test case definition within the root group. You can use these IDs to run a subset of your test suite definition.

Java SDK example:

```
response = iotDeviceAdvisorClient.GetSuiteDefinition(
    GetSuiteDefinitionRequest.builder()
        .suiteDefinitionId("your-suite-definition-id")
        .build()
)
```

Get a test endpoint

Use the `GetEndpoint` API operation to get the test endpoint used by your device. Select the endpoint that best fits your test. To simultaneously run multiple test suites, use the Device-level endpoint by providing a `thing ARN`, `certificate ARN`, or `device role ARN`. To run a single test suite, provide no arguments to the GetEndpoint operation to choose the Account-level endpoint.

SDK example:

```
response = iotDeviceAdvisorClient.getEndpoint(GetEndpointRequest.builder()
.certificateArn("your-test-device-certificate-arn")
.thingArn("your-test-device-thing-arn")
.deviceRoleArn("your-device-role-arn") //if using SigV4 for MQTT over WebSocket

.build())
```

Start a test suite run

After you create a test suite definition and configureyour test device to connect to your Device Advisor test endpoint, run your test suite with the `StartSuiteRun` API.

For MQTT customers, use either `certificateArn` or `thingArn` to run the test suite. If both are configured, the certificate is used if it belongs to the thing.

For MQTT over WebSocket customer, use `deviceRoleArn` to run the test suite. If the specified role is different from the role specified in the test suite definition, the specified role overrides the defined role.

For `.parallelRun()`, use `true` if you use a Device-level endpoint to run multiple test suites in parallel using one AWS account.

SDK example:

```
response = iotDeviceAdvisorClient.startSuiteRun(StartSuiteRunRequest.builder()
.suiteDefinitionId("your-suite-definition-id")
.suiteRunConfiguration(SuiteRunConfiguration.builder()
    .primaryDevice(DeviceUnderTest.builder()
        .certificateArn("your-test-device-certificate-arn")
        .thingArn("your-test-device-thing-arn")
        .deviceRoleArn("your-device-role-arn") //if using SigV4 for MQTT over WebSocket

        .build())
    .parallelRun(true | false)
    .build())
.build())
```

Save the `suiteRunId` from the response. You will use this to retrieve the results of this test suite run.

Get a test suite run

After you start a test suite run, you can check its progress and its results with the `GetSuiteRun` API.

SDK example:

```
// Using the SDK, call the GetSuiteRun API.

response = iotDeviceAdvisorClient.GetSuiteRun(
GetSuiteRunRequest.builder()
    .suiteDefinitionId("your-suite-definition-id")
    .suiteRunId("your-suite-run-id")
.build())
```

Stop a test suite run

To stop a test suite run that is still in progress, you can call the `StopSuiteRun` API operation. After you call the `StopSuiteRun` operation, the service starts the cleanup process. While the service runs the cleanup process, the test suite run status updates to `Stopping`. The cleanup process can take several minutes. Once the process is complete, the test suite run status updates to `Stopped`. After a test run has completely stopped, you n start another test suite run. You can periodically check the suite run status using the `GetSuiteRun` API operation, as shown in the previous section.

SDK example:

```
// Using the SDK, call the StopSuiteRun API.

response = iotDeviceAdvisorClient.StopSuiteRun(
StopSuiteRun.builder()
    .suiteDefinitionId("your-suite-definition-id")
    .suiteRunId("your-suite-run-id")
.build())
```

Get a qualification report for a successful qualification test suite run

If you run a qualification test suite that completes successfully, you can retrieve a qualification report with the `GetSuiteRunReport` API operation. You use this qualification report to qualify your device with the AWS IoT Core qualification program. To determine whether your test suite is a qualification test suite, check whether the `intendedForQualification` parameter is set to `true`. After you call the `GetSuiteRunReport` API operation, you can download the report from the returned URL for up to 90 seconds. If more than 90 seconds elapse from the previous time you called the `GetSuiteRunReport` operation, call the operation again to retrieve a new, valid URL.

SDK example:

```
// Using the SDK, call the getSuiteRunReport API.

response = iotDeviceAdvisorClient.getSuiteRunReport(
    GetSuiteRunReportRequest.builder()
        .suiteDefinitionId("your-suite-definition-id")
        .suiteRunId("your-suite-run-id")
        .build()
)
```

Device Advisor detailed console workflow

In this tutorial, you'll create a custom test suite and run tests against the device you want to test in the console. After the tests are complete, you can view the test results and detailed logs.

Tutorials

- Prerequisites
- Create a test suite definition
- Start a test suite run

- Stop a test suite run (optional)
- View test suite run details and logs
- Download an AWS IoT qualification report

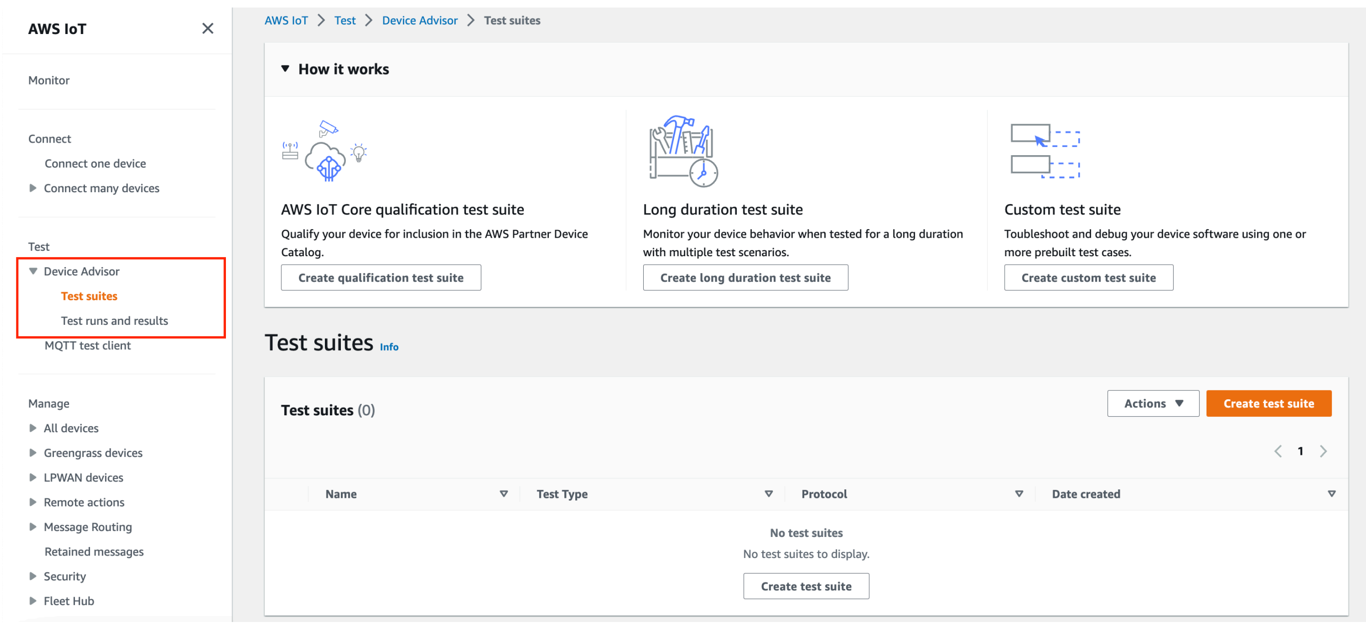
Prerequisites

To complete this tutorial, you need to create a thing and certificate.

Create a test suite definition

Create a test suite suite so that you can run it for your devices and perform verification.

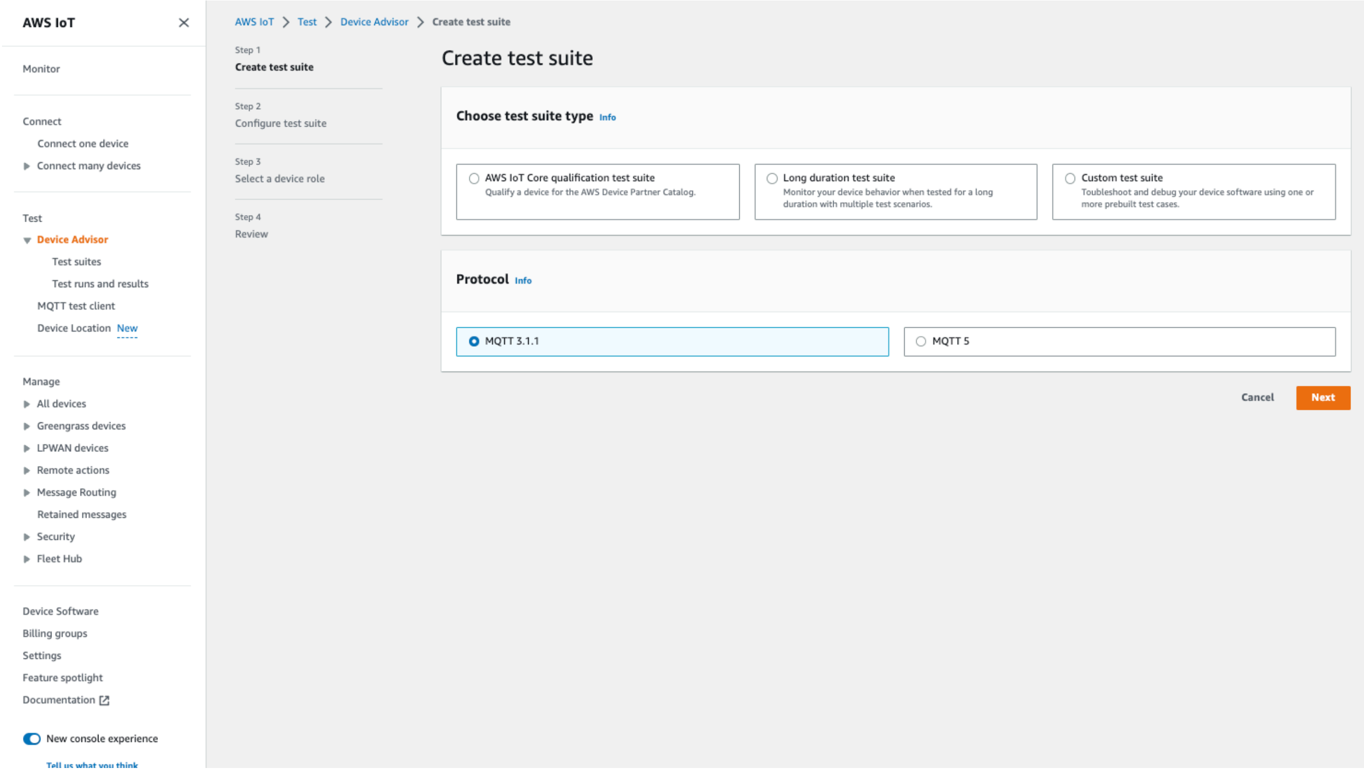
1. In the AWS IoT console, in the navigation pane, expand **Test**, **Device Advisor** and then choose **Test suites**.



   Choose **Create Test Suite**.

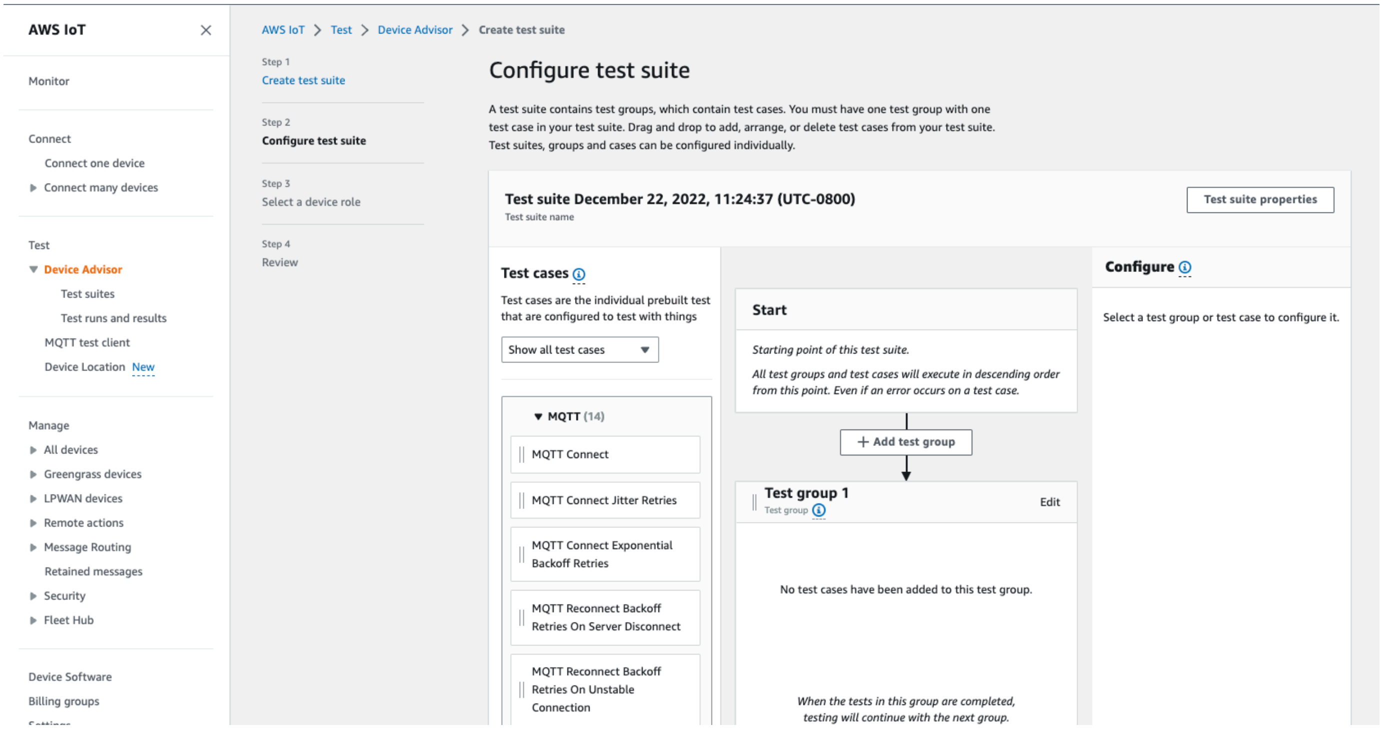
2. Select either `Use the AWS Qualification test suite` or `Create a new test suite`.

   For protocol, choose either **MQTT 3.1.1** or **MQTT 5**.

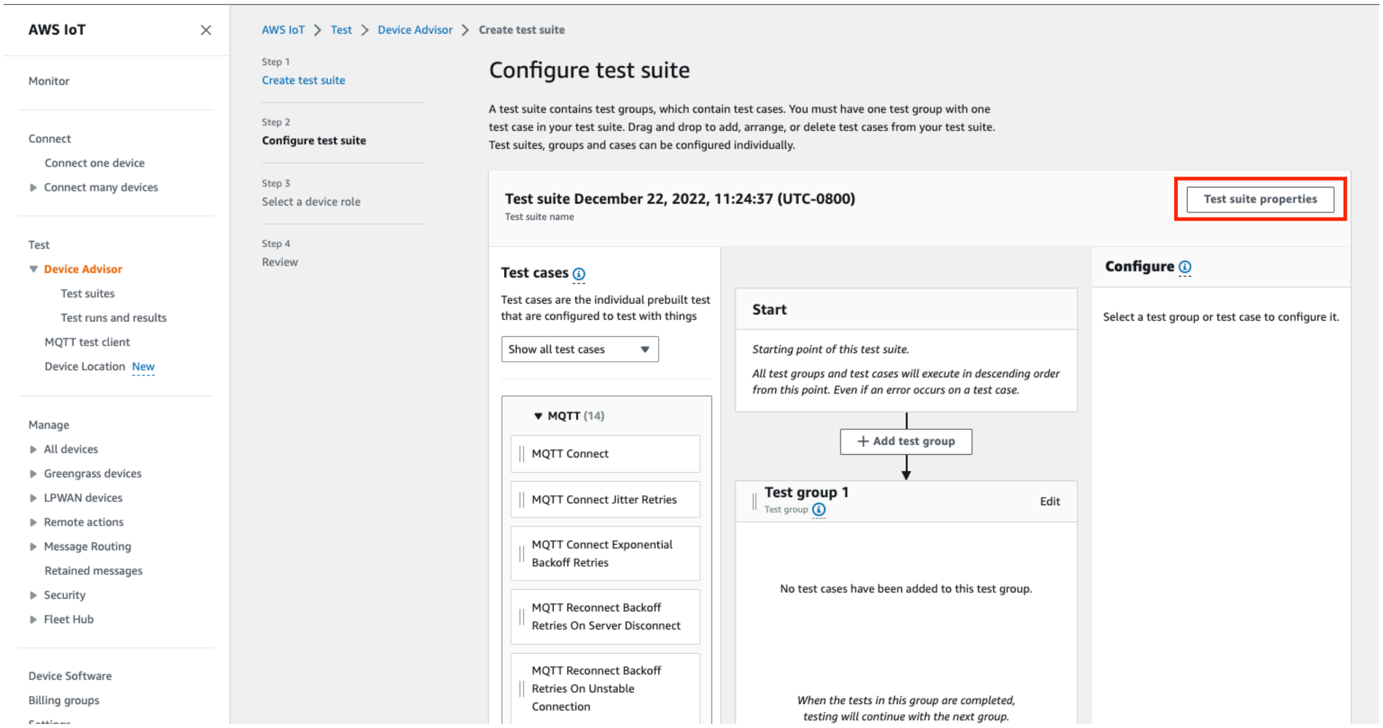


Select `Use the AWS Qualification test suite` to qualify and list your device to the AWS Partner Device Catalog. By choosing this option, test cases required for qualification of your device to the AWS IoT Core qualification program are pre-selected. Test groups and test cases can't be added or removed. You will still need to configure the test suite properties.

Select `Create a new test suite` to create and configure a custom test suite. We recommend starting with this option for initial testing and troubleshooting. A custom test suite must have at least one test group, and each test group must have at least one test case. For the purpose of this tutorial, we'll select this option and choose **Next**.



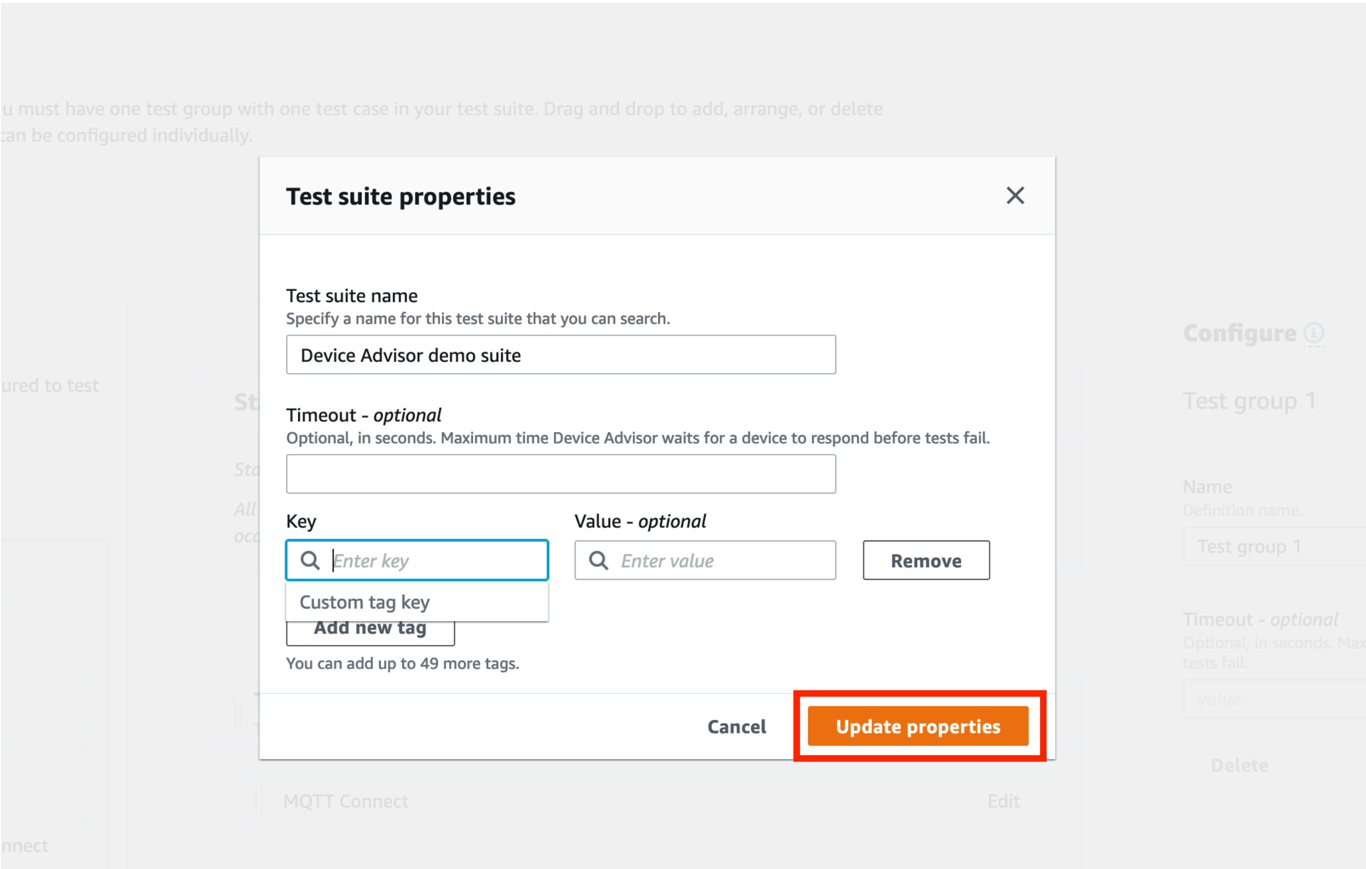
3. Choose **Test suite properties**. You must create the test suite properties when you create your test suite.



Under **Test suite properties**, fill out the following:

- **Test suite name**: You can create the suite with a custom name.

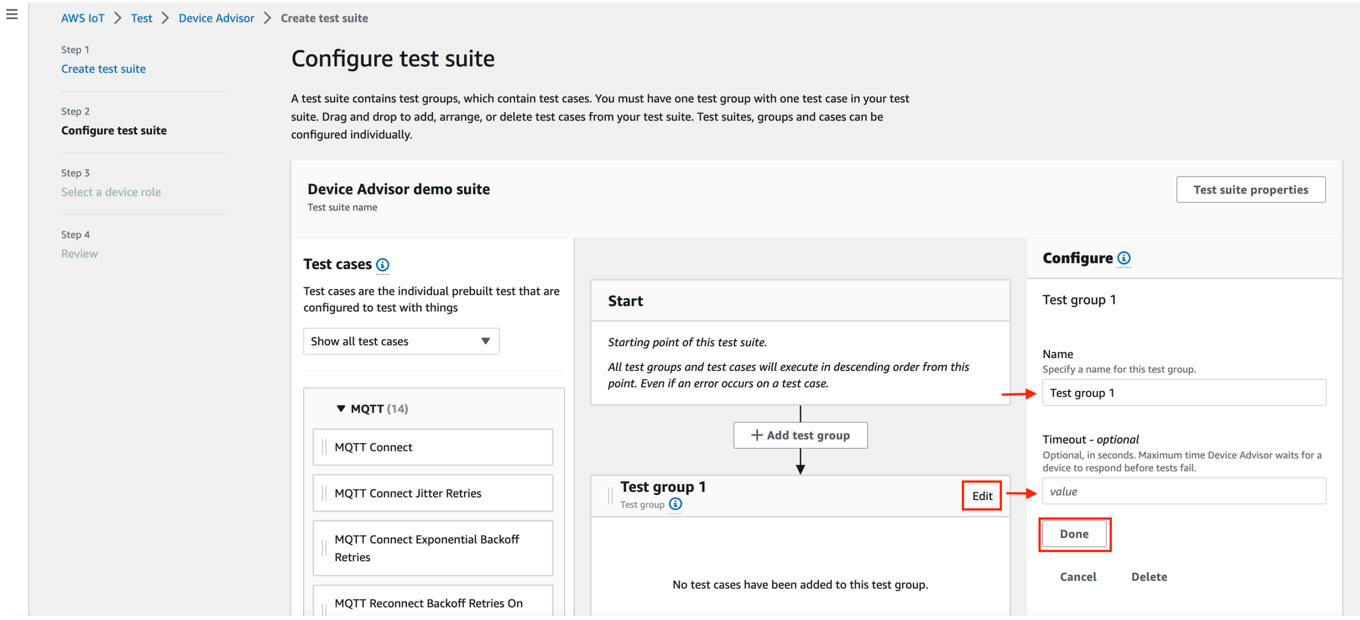
- **Timeout** (optional): The timeout in seconds for each test case in the current test suite. If you don't specify a timeout value, the default value is used.
- **Tags** (optional): Add tags to the test suite.



When you've finished, choose **Update properties**.

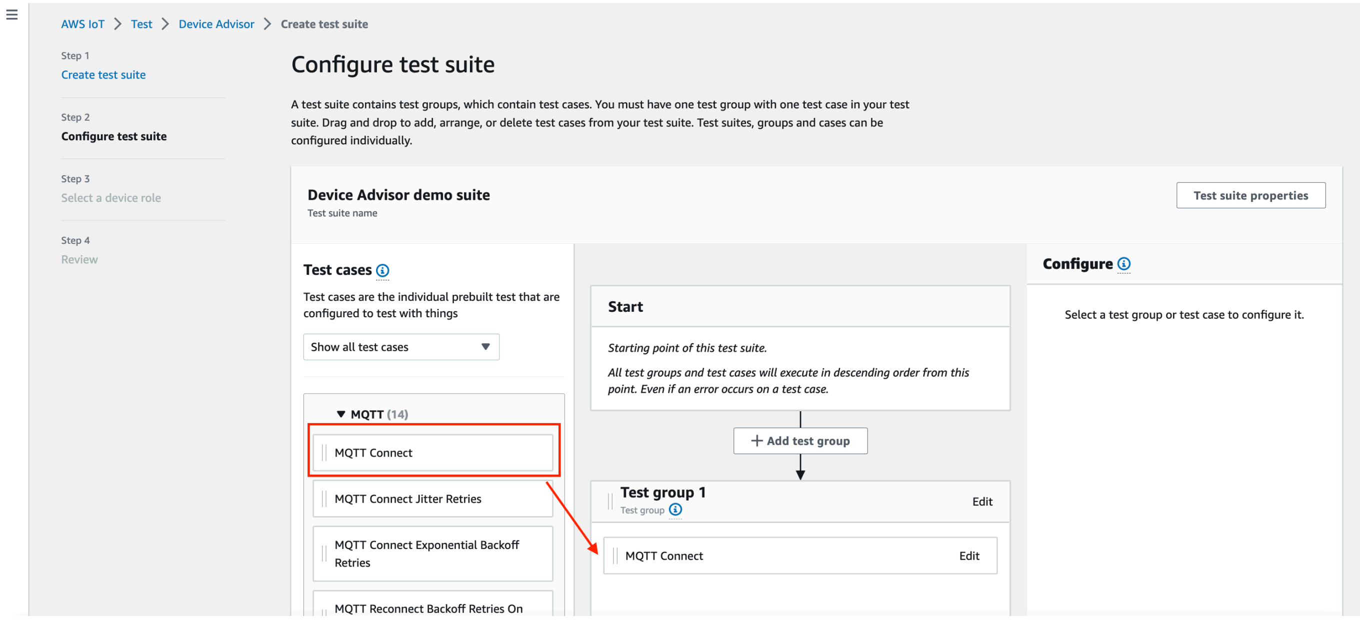
4. To modify the group level configuration, under `Test group 1`, choose **Edit**. Then, enter a **Name** to give the group a custom name.

   Optionally, you can also enter a **Timeout** value in seconds under the selected test group. If you don't specify a timeout value, the default value is used.



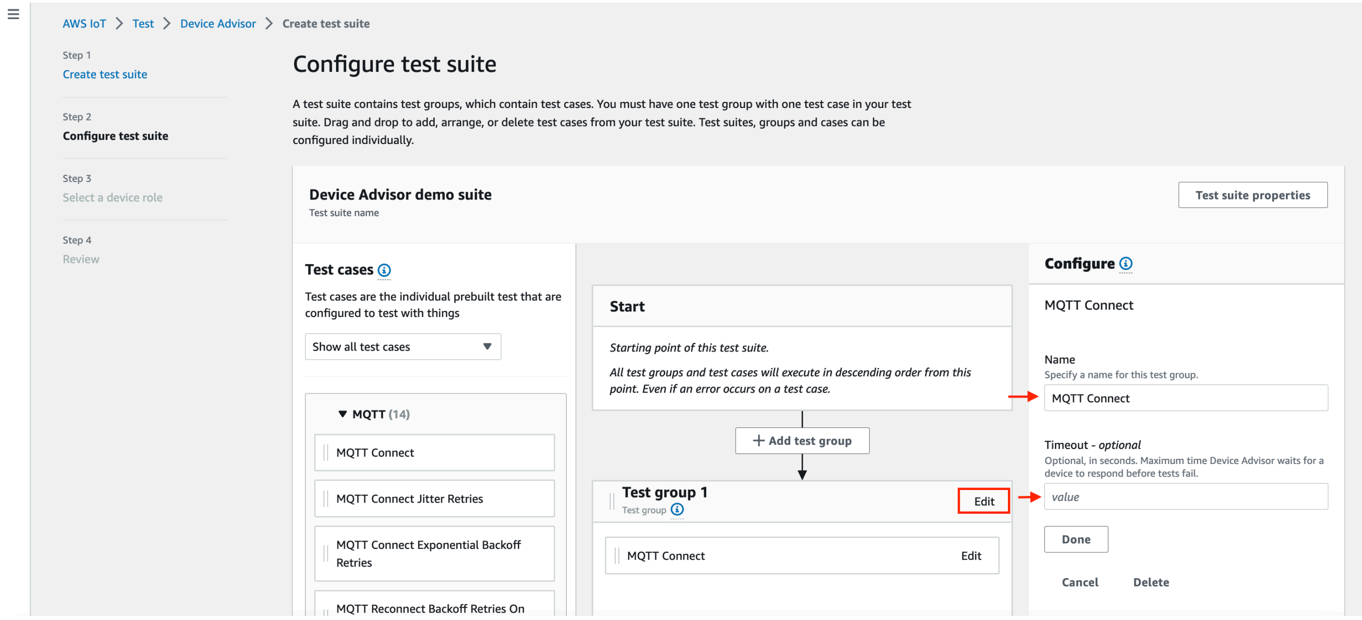
Choose **Done**.

5. Drag one of the available test cases from **Test cases** into the test group.



6. To modify the test case level configuration for the test case that you added to your test group, choose **Edit**. Then, enter a **Name** to give the group a custom name.

   Optionally, you can also enter a **Timeout** value in seconds under the selected test group. If you don't specify a timeout value, the default value is used.



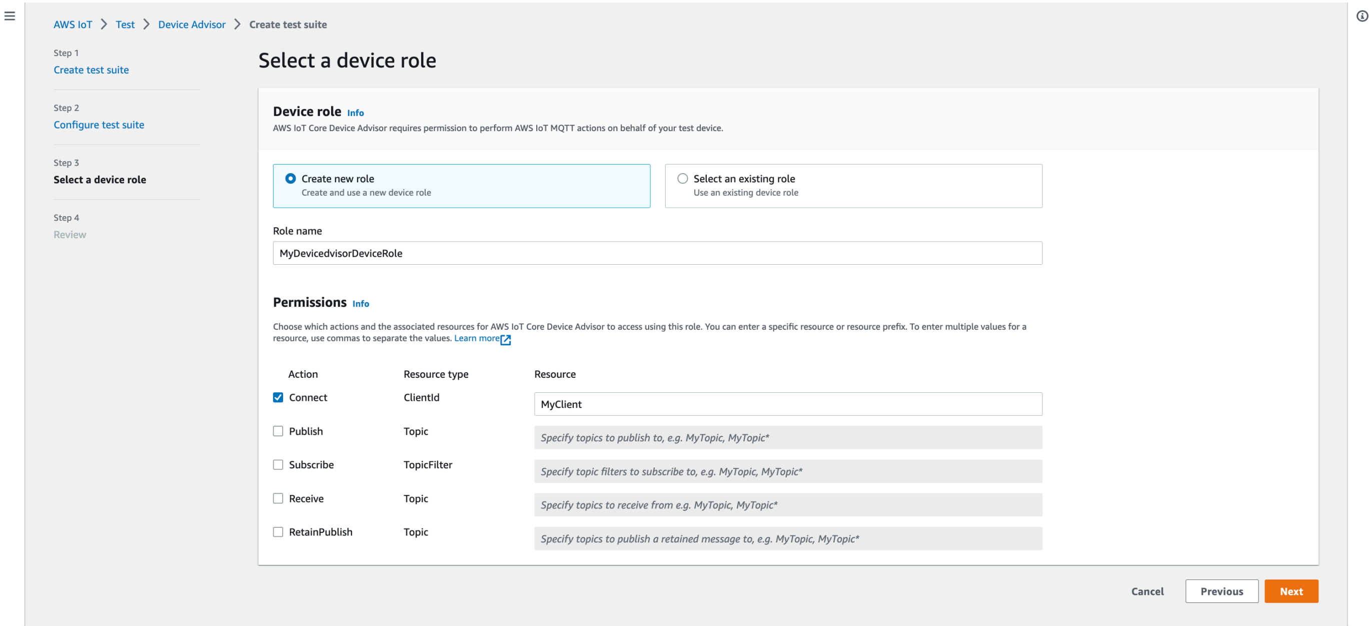
Choose **Done**.

> **Note**
>
> To add more test groups to the test suite, choose **Add test group**. Follow the preceding steps to create and configure more test groups, or to add more test cases to one or more test groups. Test groups and test cases can be reordered by choosing and dragging a test case to the desired position. Device Advisor runs tests in the order in which you define the test groups and test cases.

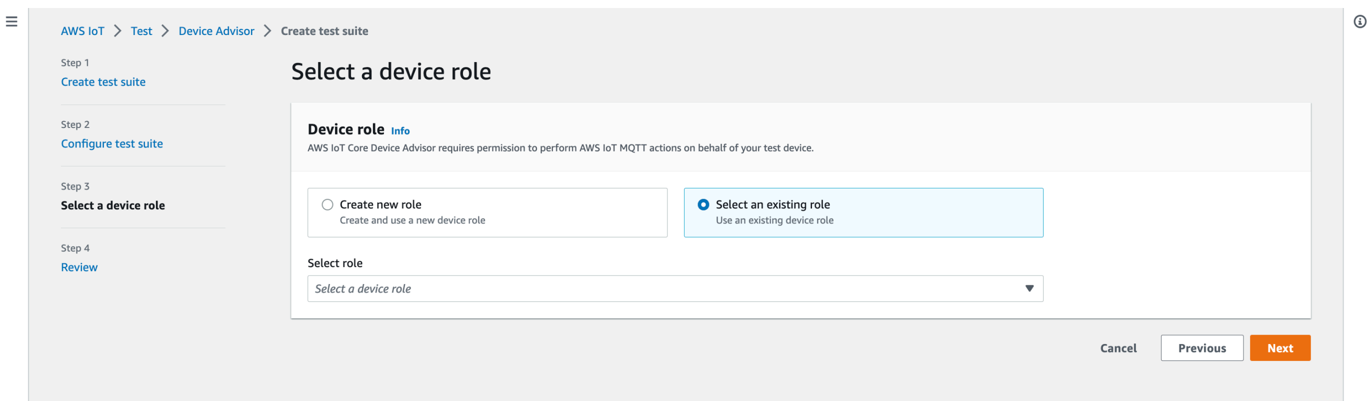
7. Choose **Next**.

8. In **Step 3**, configure a device role which Device Advisor will use to perform AWS IoT MQTT actions on behalf of your test device.

   If you selected **MQTT Connect** test case only in **Step 2**, the **Connect** action will be checked automatically since that permission is required on device role to run this test suite. If you selected other test cases, the corresponding required actions will be checked. Ensure that the resource values values for each of the actions is provided. For example, for the **Connect** action, provide the client id that your device will be connecting to the Device Advisor endpoint with. You can provide multiple values by using commas to separate the values, and you can provide prefix values using a wildcard (*) character as well. For example, to provide permission to publish on any topic beginning with `MyTopic`, you can provide "`MyTopic*`" as the resource value.



If you already created a device role previously and would like to use that role, select **Select an existing role** and choose your device role under **Select role**.



Configure your device role using one of the two provided options and choose **Next**.

9. In **Step 4**, make sure the configuration provided in each of the steps is accurate. To edit configuration provided for a particular step, choose **Edit** for the corresponding step.

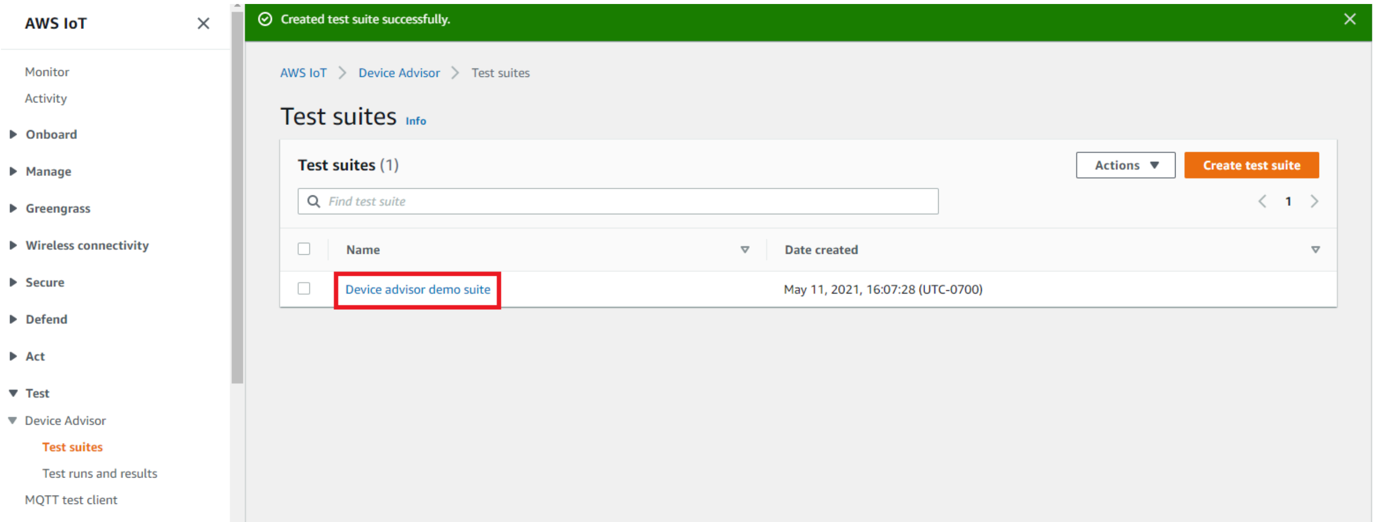
   After you verify the configuration, choose **Create test suite**.

   The test suite should be created successfully and you'll be redirected to the **Test suites** page where you can view all the test suite that have been created.

   If the test suite creation failed, make sure the test suite, test groups, test cases, and device role have been configured according to the previous instructions.

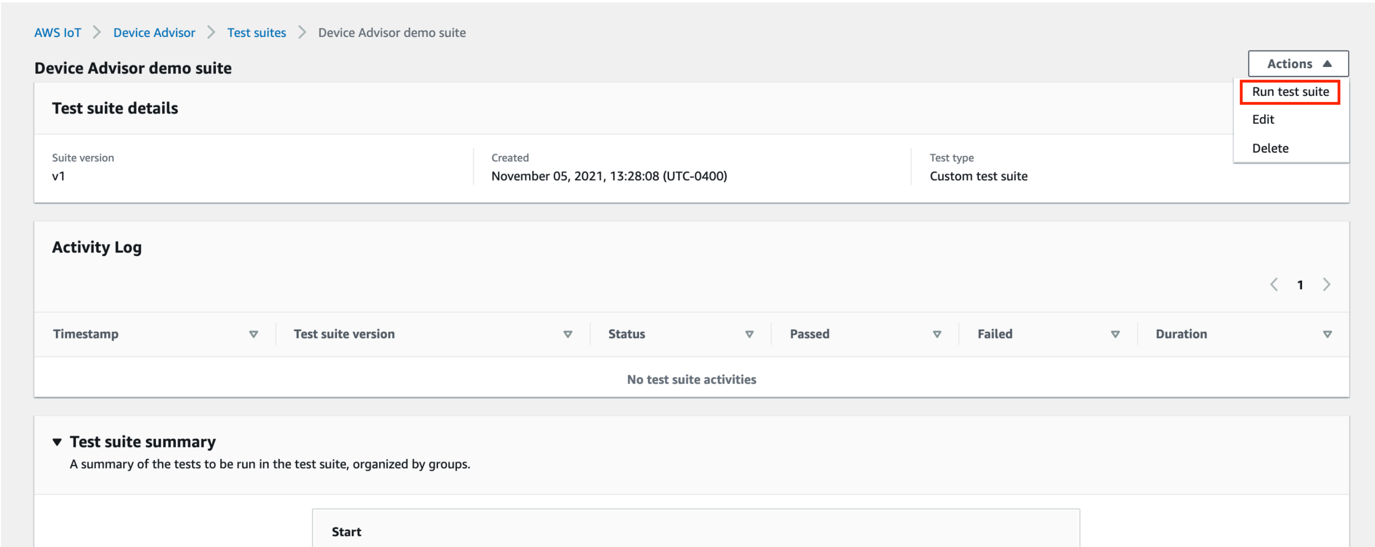
Start a test suite run

1. In the AWS IoT console, in the navigation pane, expand **Test**, **Device Advisor**, and then choose **Test suites**.
2. Choose the test suite for which you'd like to view the test suite details.



The test suite detail page displays all of the information related to the test suite.

3. Choose **Actions**, then **Run test suite**.

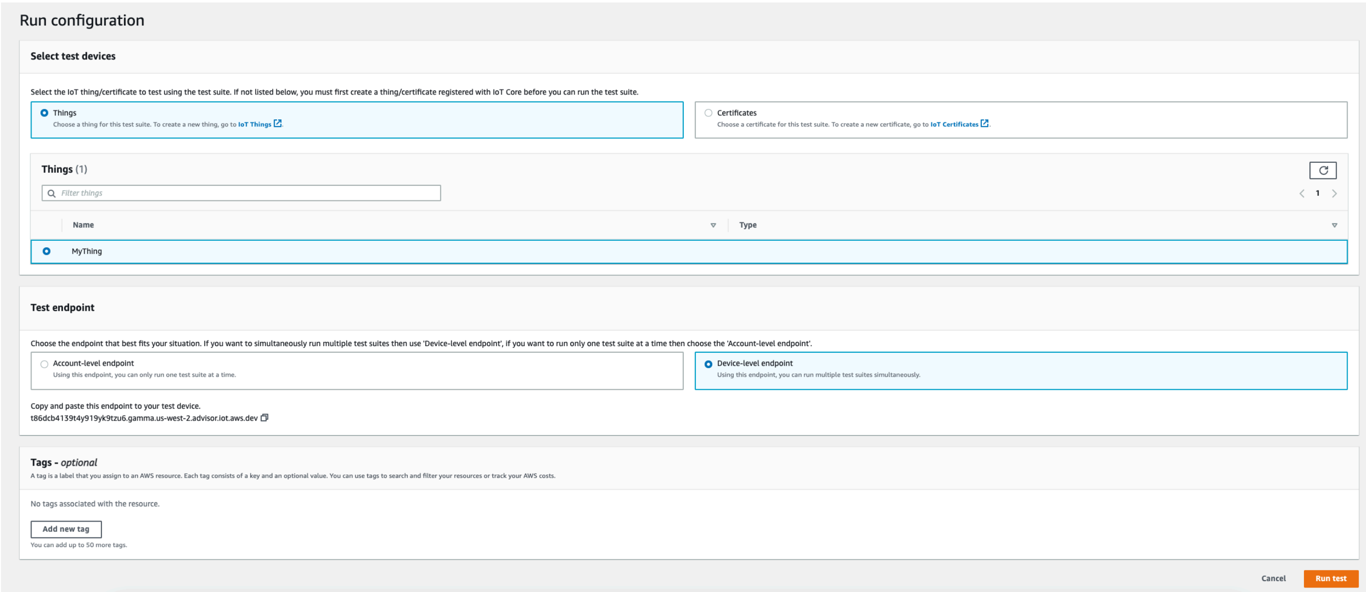


4. Under **Run configuration**, you'll need to select an AWS IoT thing or certificate to test using Device Advisor. If you don't have any existing things or certificates, first create AWS IoT Core resources.

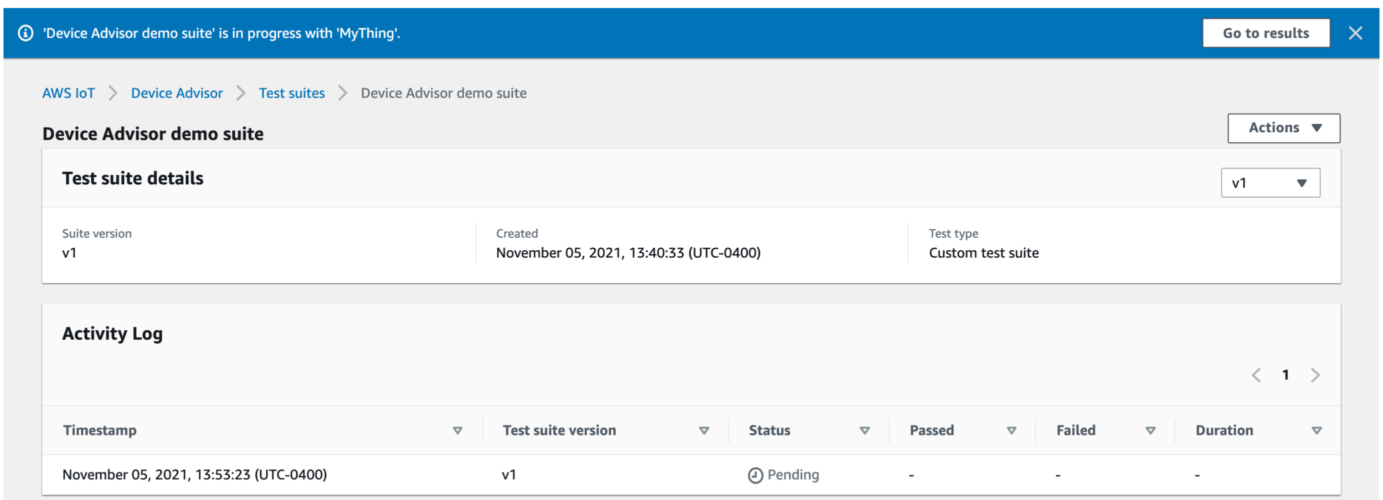
In **Test endpoint** section, select the endpoint that best fits your case. If you plan to run multiple test suites simultaneously using the same AWS account in the future, select **Device-level endpoint**. Otherwise, if you plan to only run one test suite at a time, select **Account-level endpoint**.

Configure your test device with the selected Device Advisor's test endpoint.

After you select a thing or certificate and choose a Device Advisor endpoint, choose **Run test**.

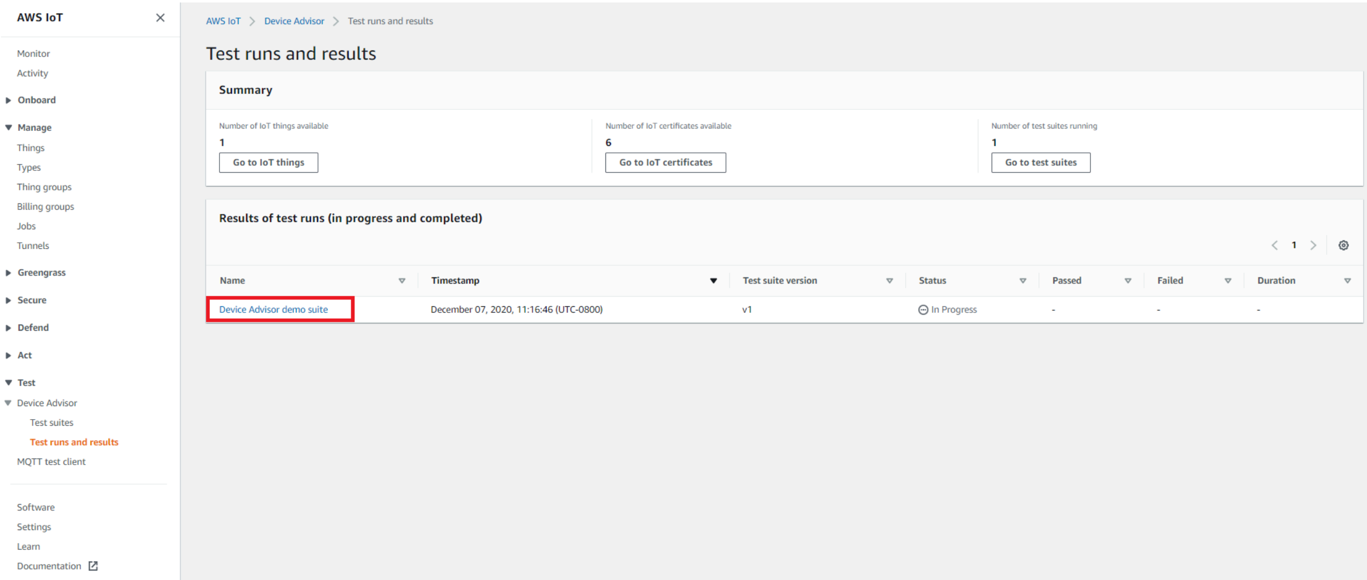


5. Choose **Go to results** on the top banner for viewing the test run details.

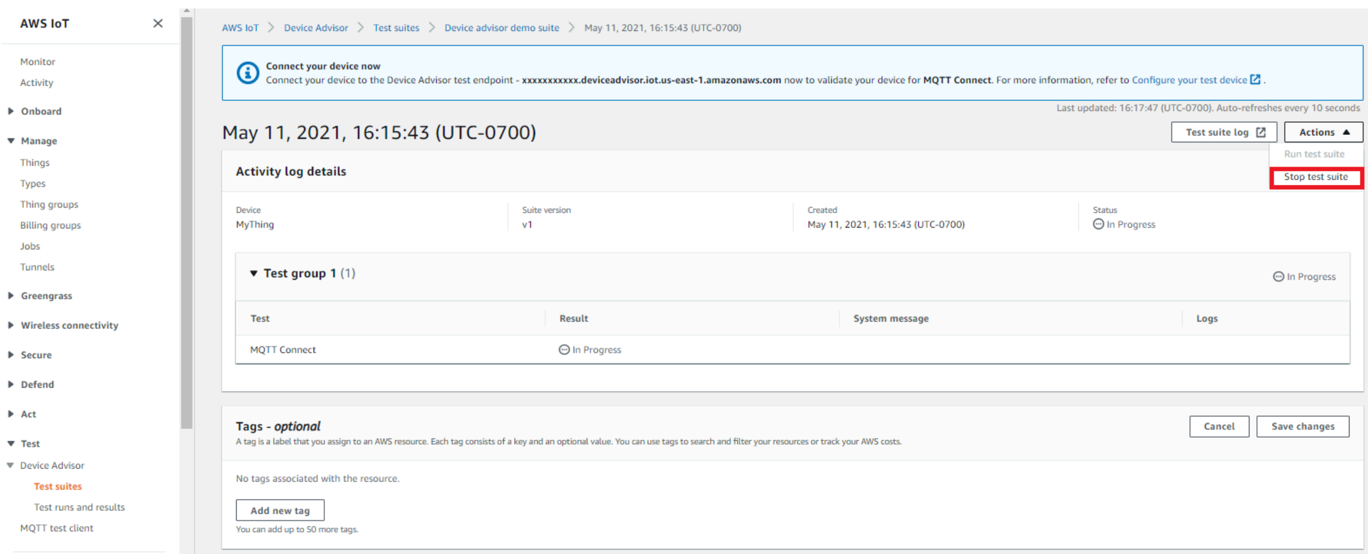


Stop a test suite run (optional)

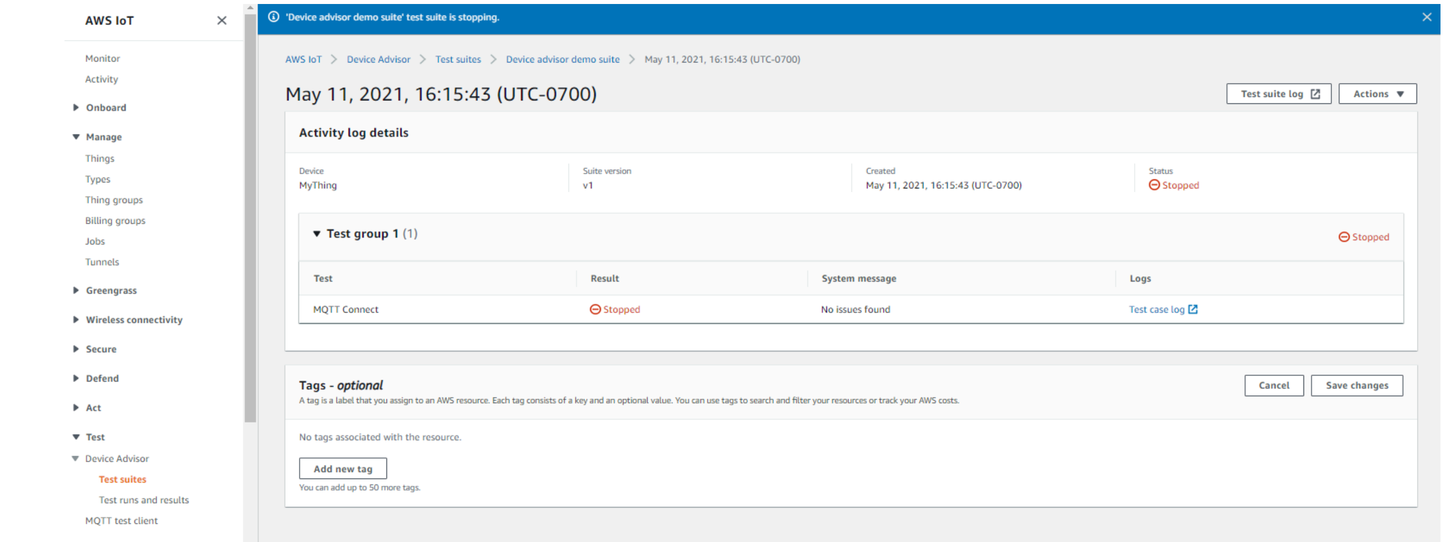
1. In the [AWS IoT console](), in the navigation pane, expand **Test**, **Device Advisor**, and then choose **Test runs and results**.
2. Choose the test suite in progress that you want to stop.



3. Choose **Actions**, then **Stop test suite**.



4. The cleanup process will take several minutes to complete. While the cleanup process runs, the test run status will be `STOPPING`. Wait for the cleanup process to complete and for the test suite status to change to the `STOPPED` status before starting a new suite run.



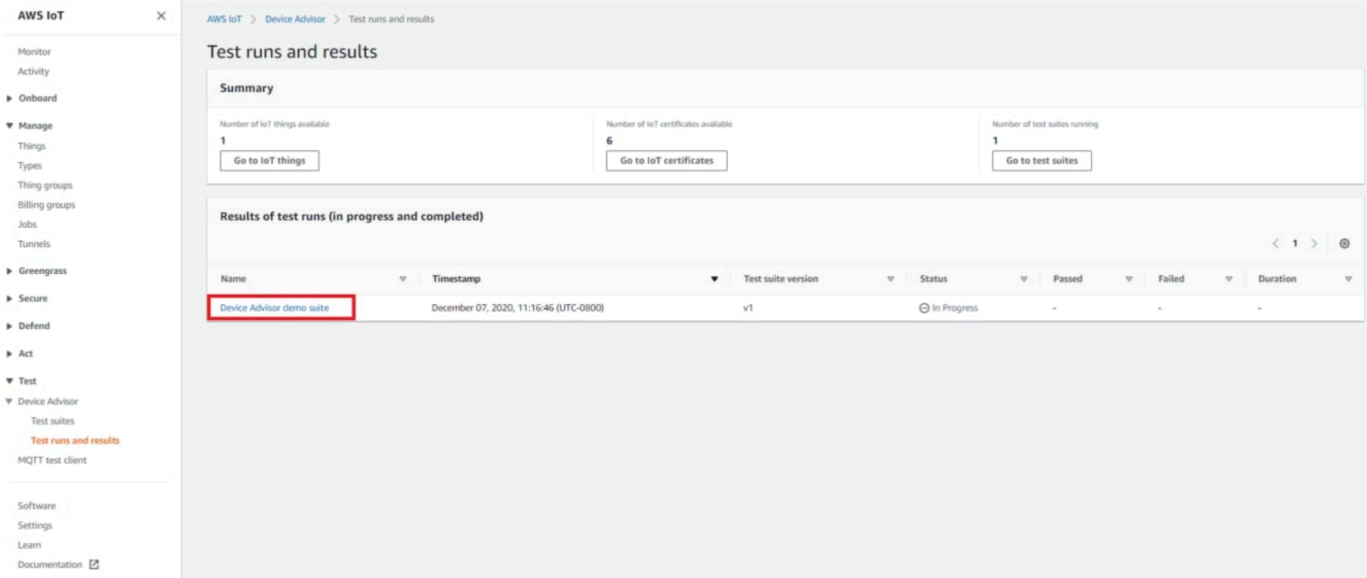
View test suite run details and logs

1. In the AWS IoT console, in the navigation pane, expand **Test**, **Device Advisor** and then choose **Test runs and results**.

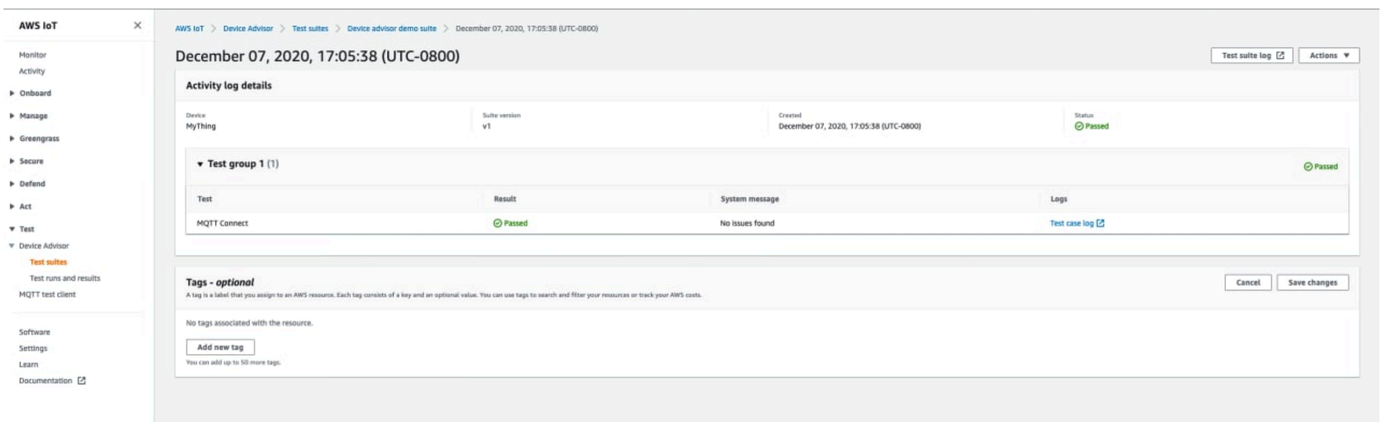
   This page displays:

   - Number of IoT things
   - Number of IoT certificates
   - Number of test suites currently running
   - All the test suite runs that have been created

2. Choose the test suite for which you'd like to view the run details and logs.



The run summary page displays the status of the current test suite run. This page automatically refreshes every 10 seconds. We recommend that you have a mechanism built for your device to try connecting to our test endpoint every five seconds for one to two minutes. Then you can run multiple test cases in sequence in an automated manner.

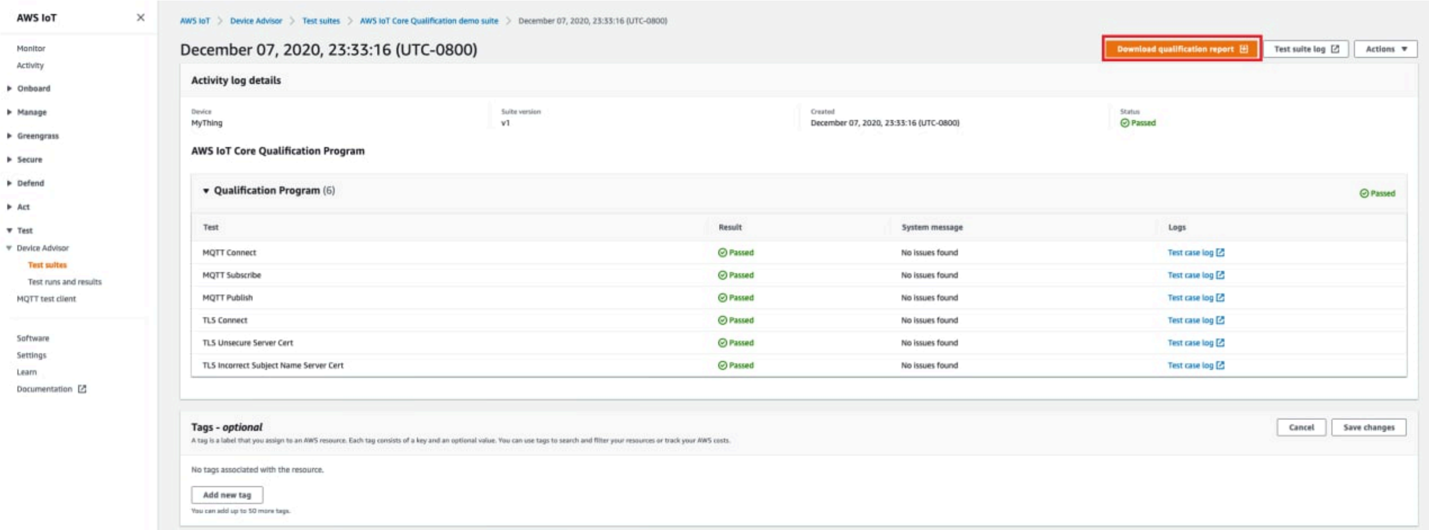


3. To access the CloudWatch logs for the test suite run, choose **Test suite log**.

   To access CloudWatch logs for any test case, choose **Test case log**.
4. Based on your test results, troubleshoot your device until all tests pass.

Download an AWS IoT qualification report

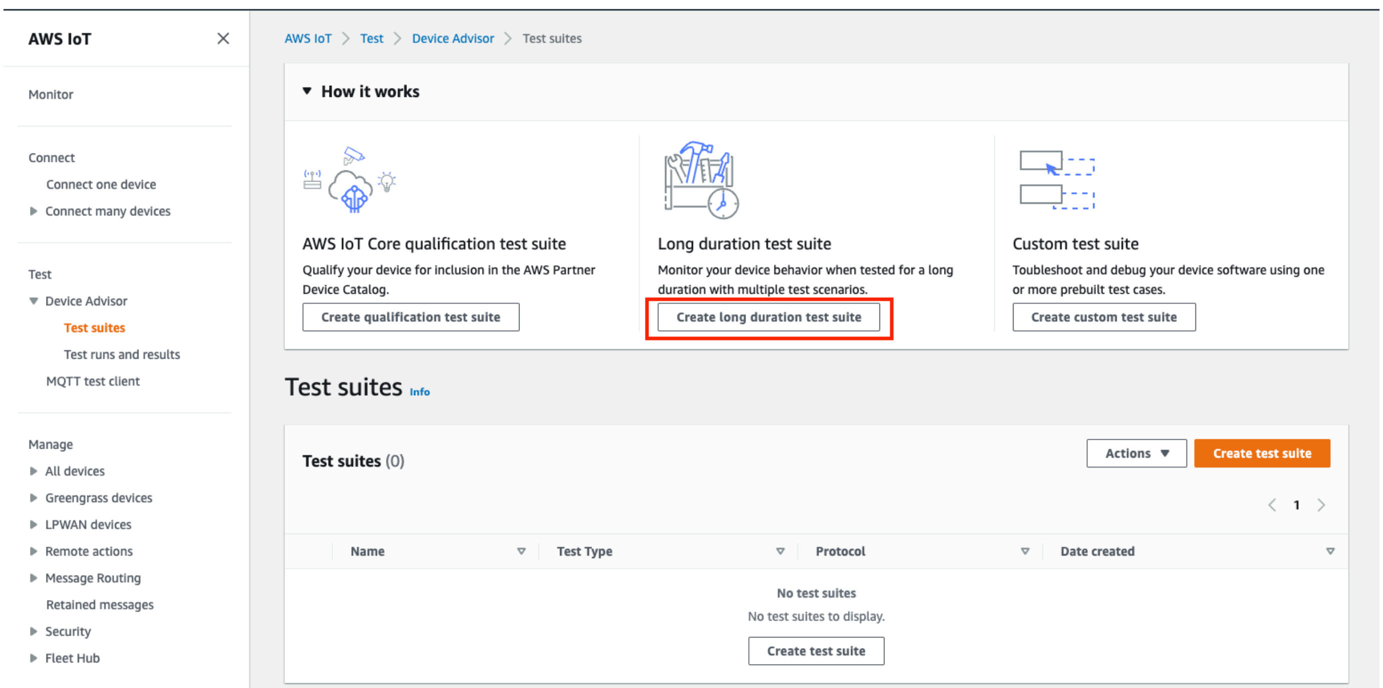
If you chose the **Use the AWS IoT Qualification test suite** option while creating a test suite and were able to run a qualification test suite, you can download a qualification report by choosing **Download qualification report** in the test run summary page.



Long duration tests console workflow

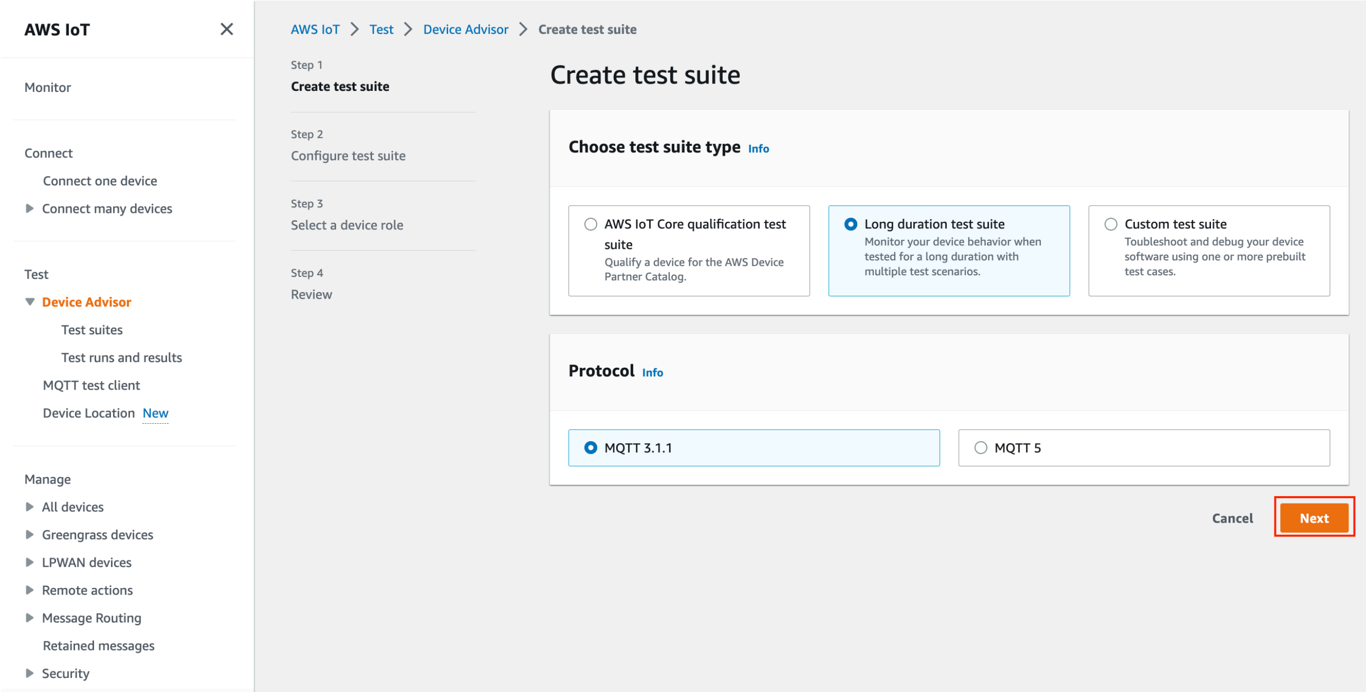
This tutorial helps you get started with the Long duration tests on Device Advisor using the console. To complete the tutorial, follow the steps at Setting up.

1. In the AWS IoT console navigation pane, expand **Test**, then **Device Advisor**, then **Test suites**. On the page, select **Create long duration test suite**.



2. On the **Create test suite** page, select **Long duration test suite** and choose **Next**.

   For protocol, choose either **MQTT 3.1.1** or **MQTT 5**.



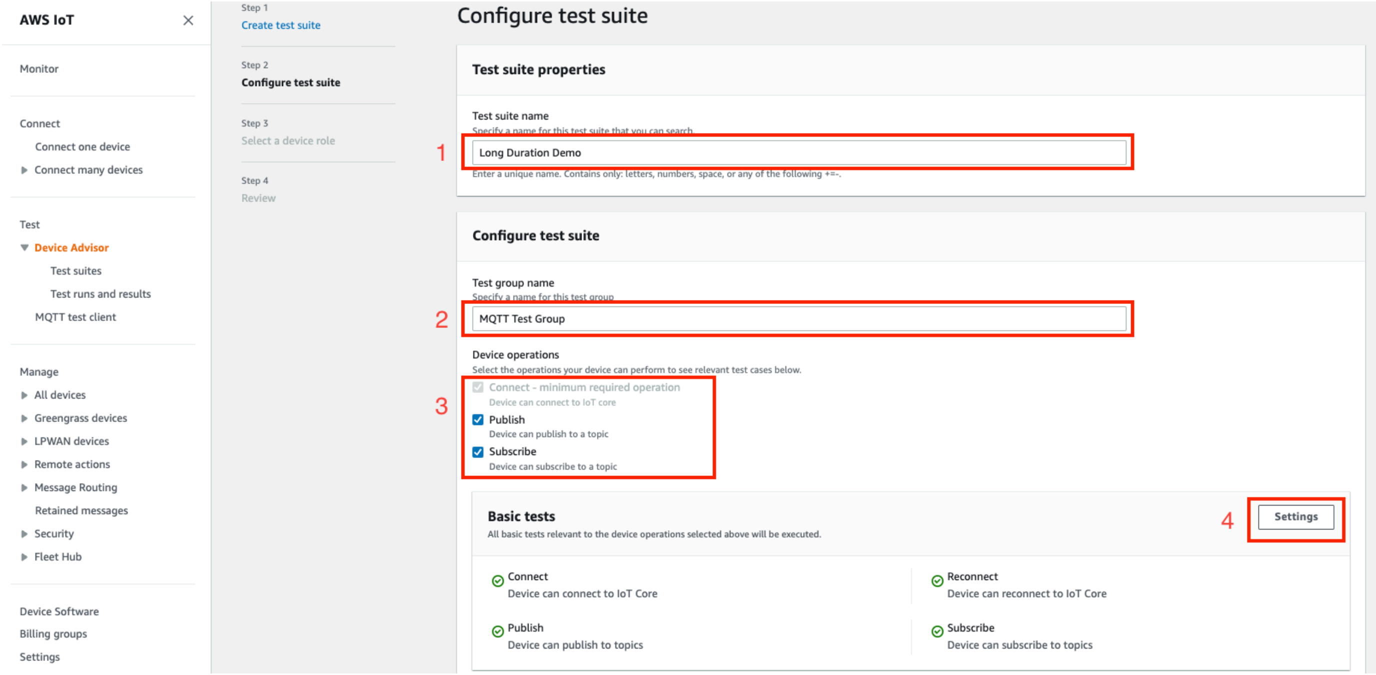
3. Do the following on the **Configure test suite** page:

   a. Update the **Test suite name** field.

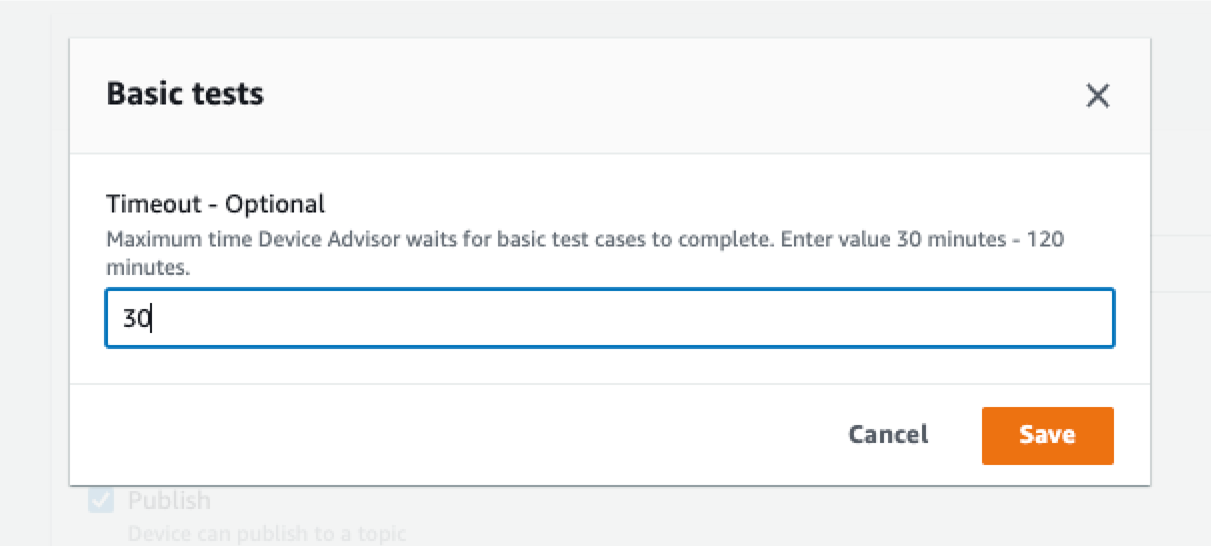
   b. Update the **Test group name** field.

   c. Choose the **Device operations** the device can perform. This will select the tests to run.

   d. Select the **Settings** option.



4. (Optional) Input the maximum amount of time Device Advisor must wait for the basic tests to complete. Select **Save**.



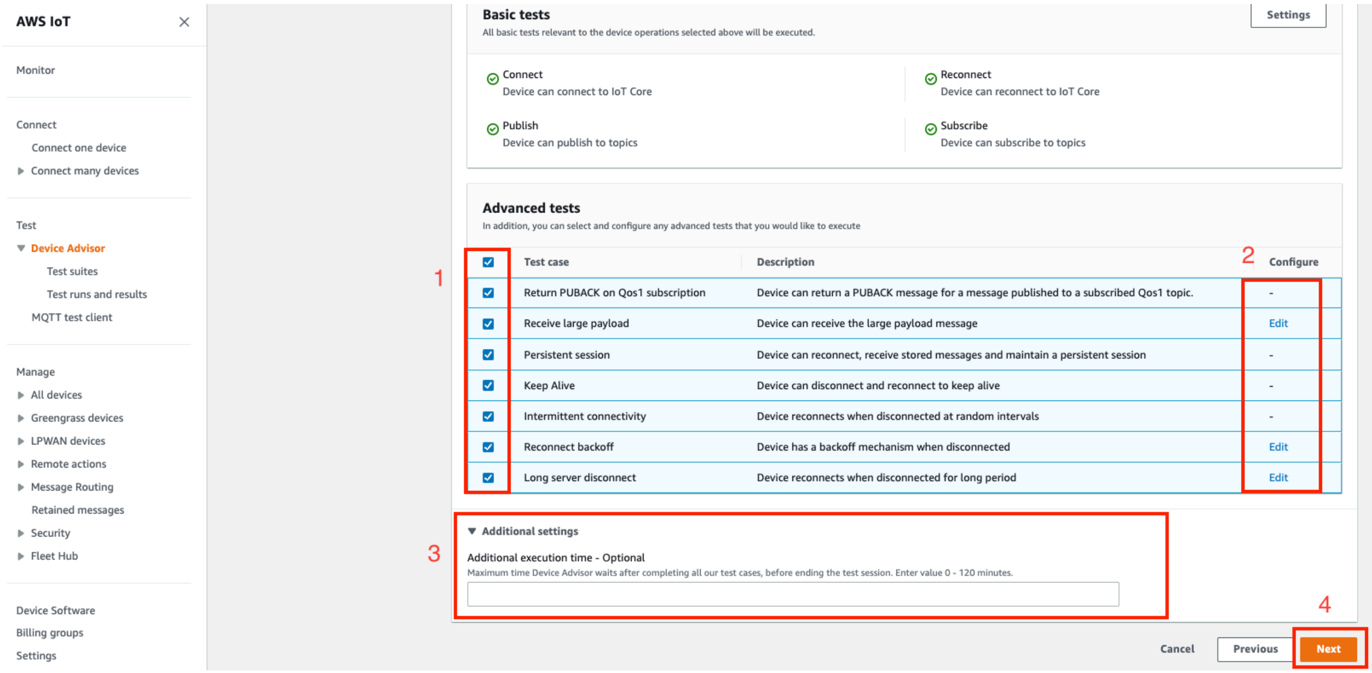
5. Do the following in the **Advanced tests** and **Additional settings** sections.

   a. Select or deselect the **Advanced tests** you want to run as part of this test.

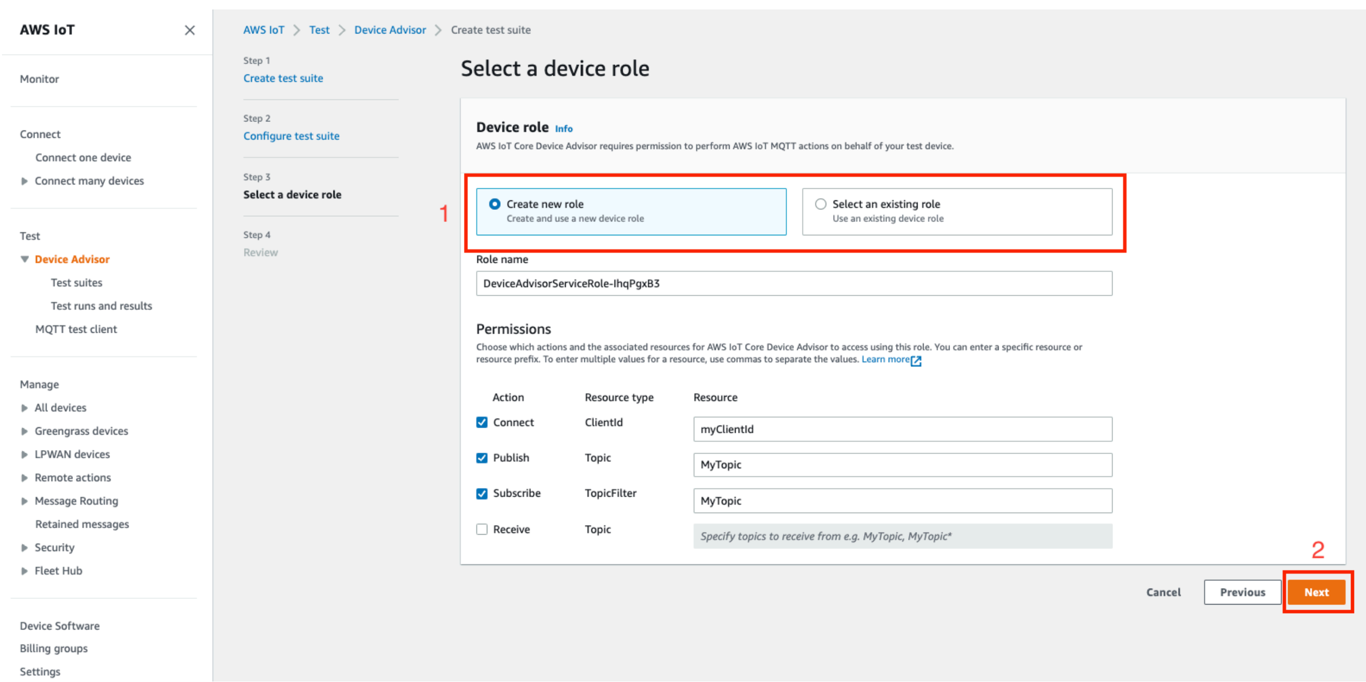
   b. **Edit** the configurations for the tests when applicable.

   c. Configure the **Additional execution time** under the **Additional settings** section.

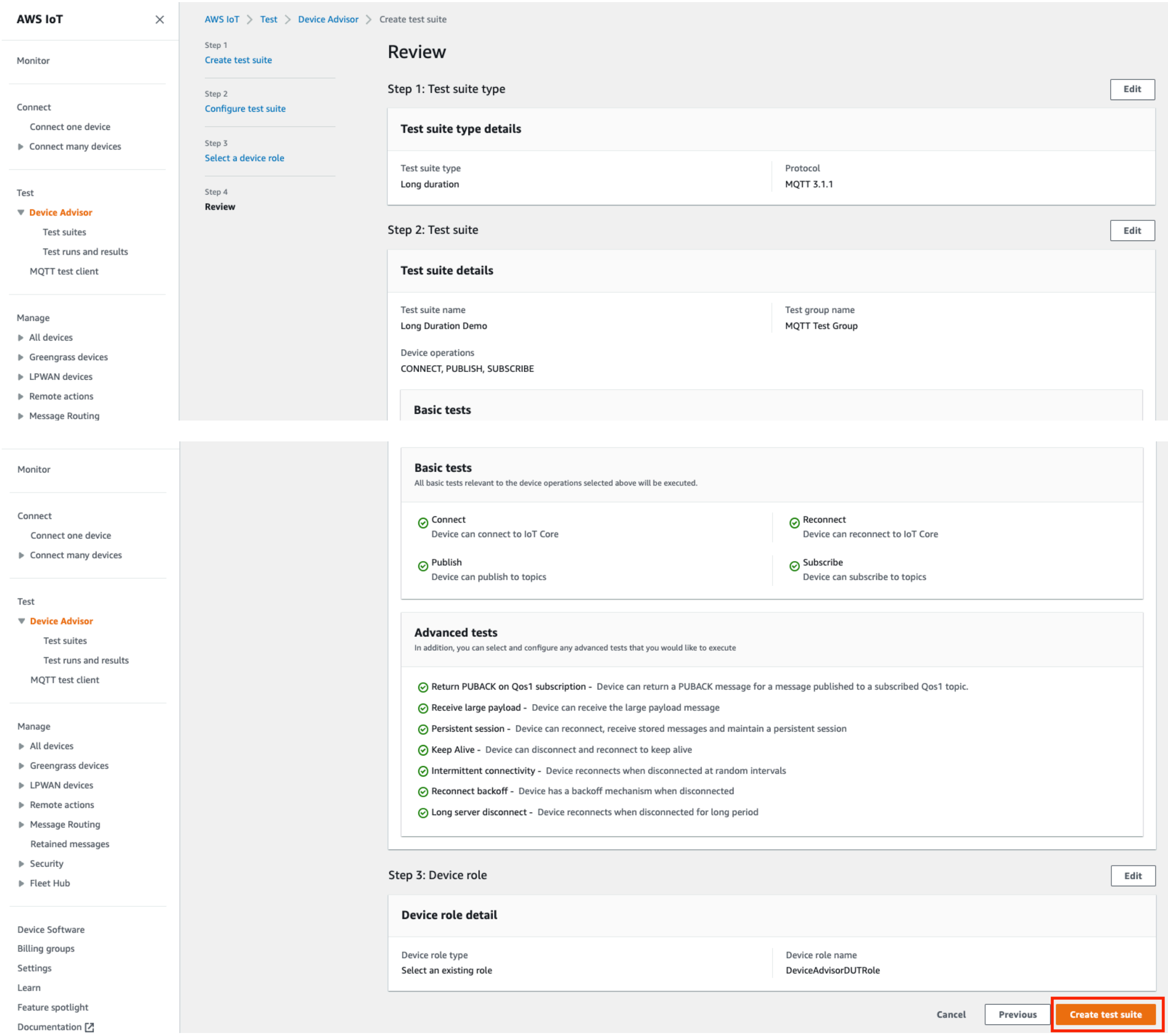
   d. Choose **Next** to do the next step.



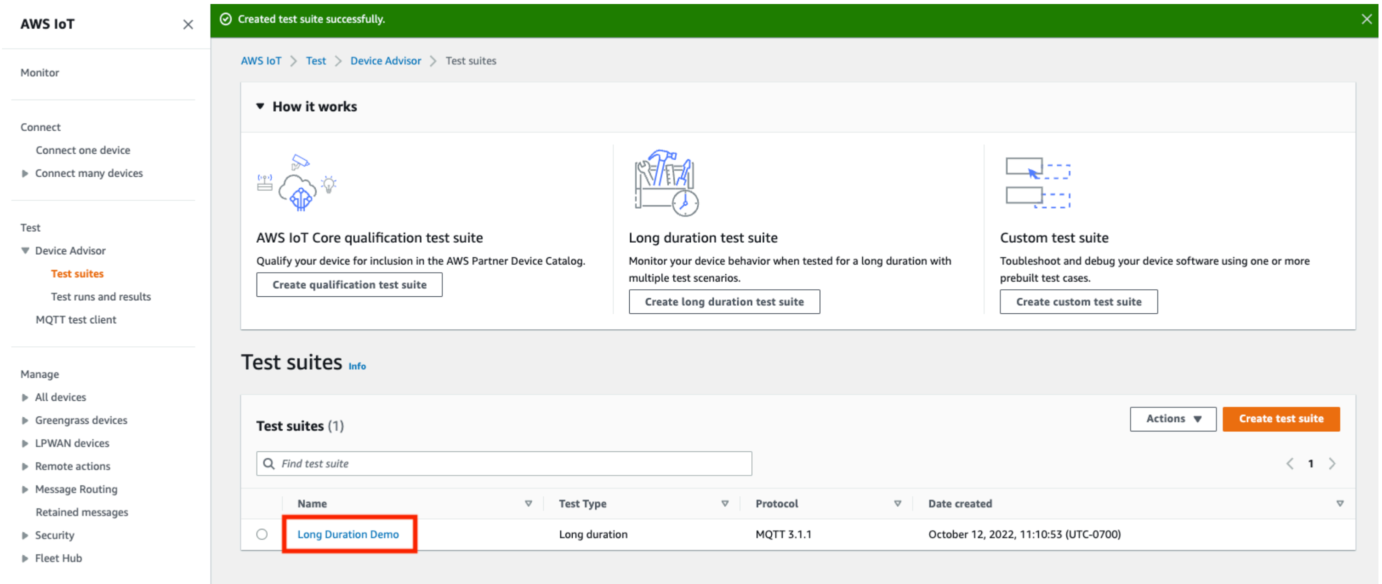
6. In this step, **Create a new role** or **Select an existing role**. See [Create an IAM role to use as your device role](#) for details.



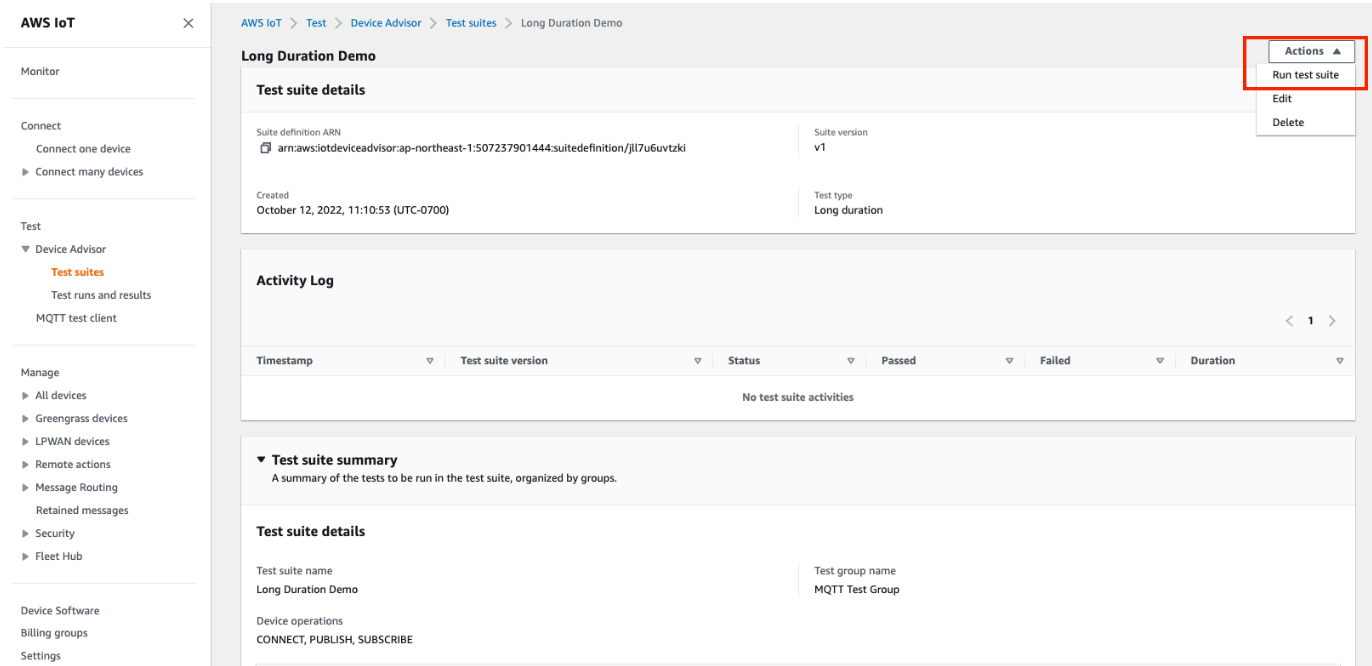
7. Review all the configurations created until this step and select **Create test suite**.



8. The created test suite is under the **Test suites** section. Select the suite to view details.



9. To run the created test suite, select **Actions** then **Run test suite**.

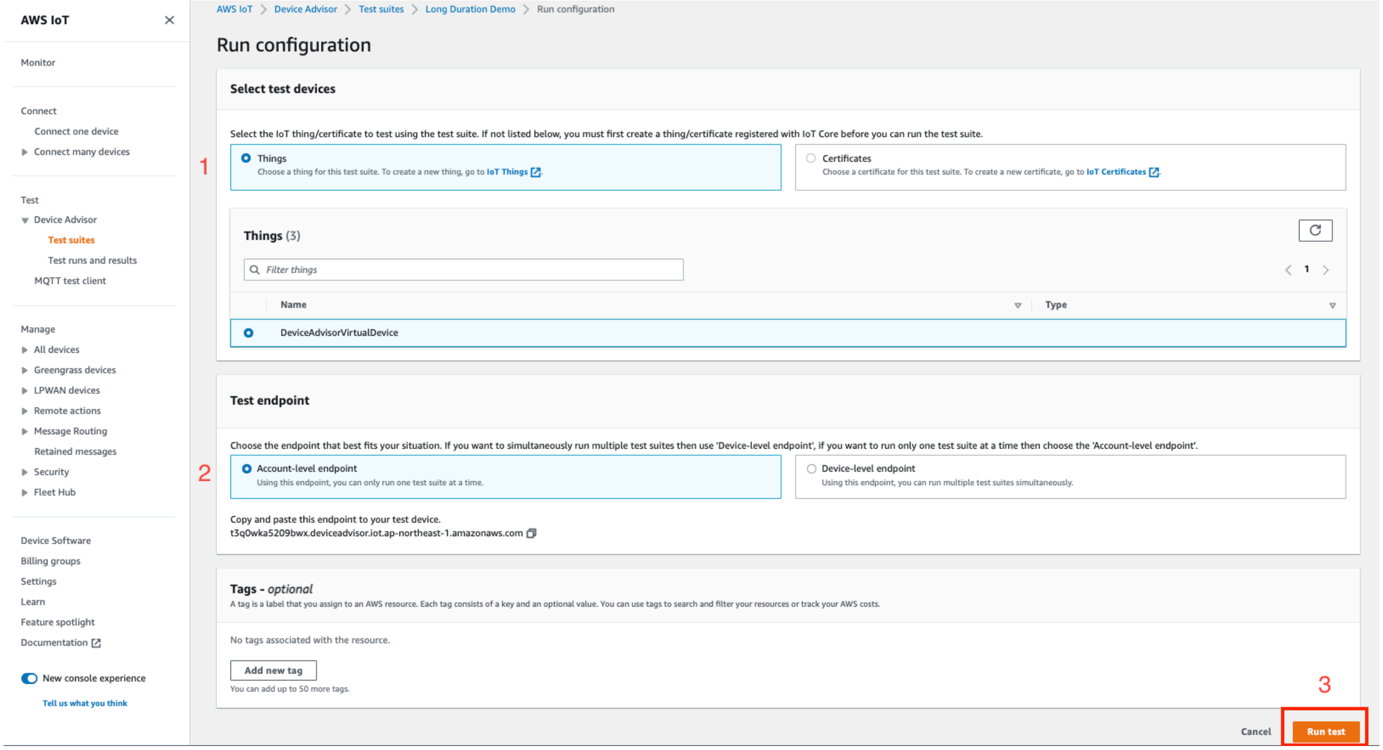


10. Choose the configuration options in the **Run configuration** page.

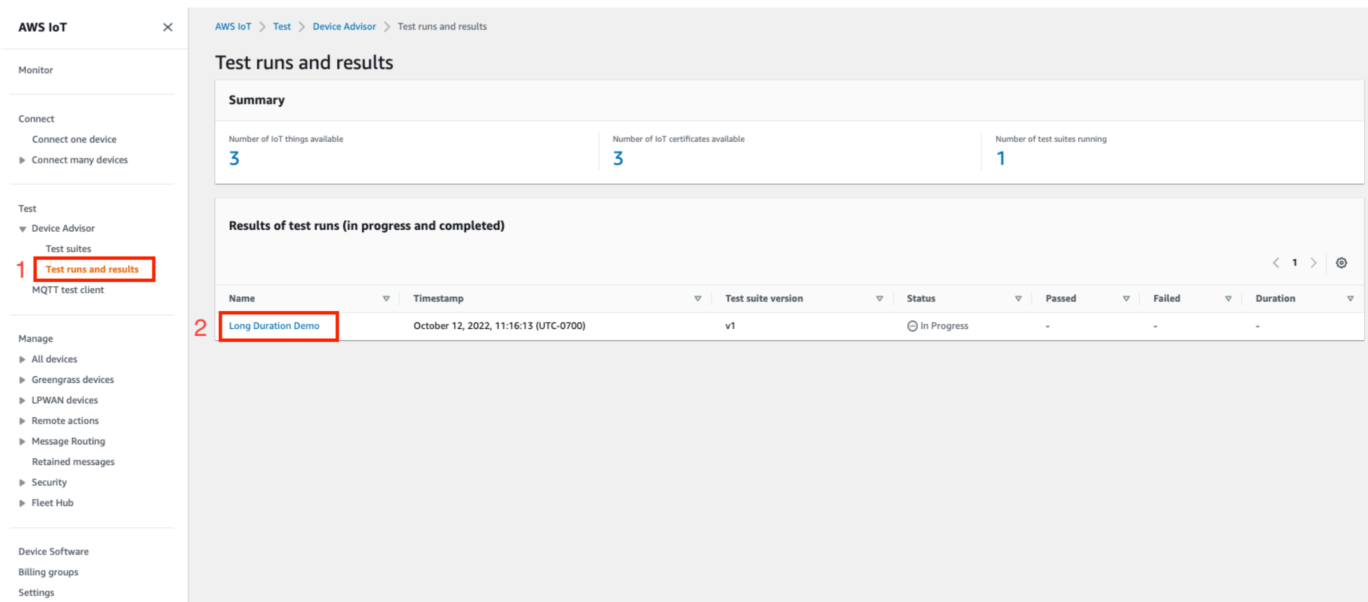
    a. Select the **Things** or **Certificate** to run the test on.

    b. Select either the **Account-level endpoint** or **Device-level endpoint**.

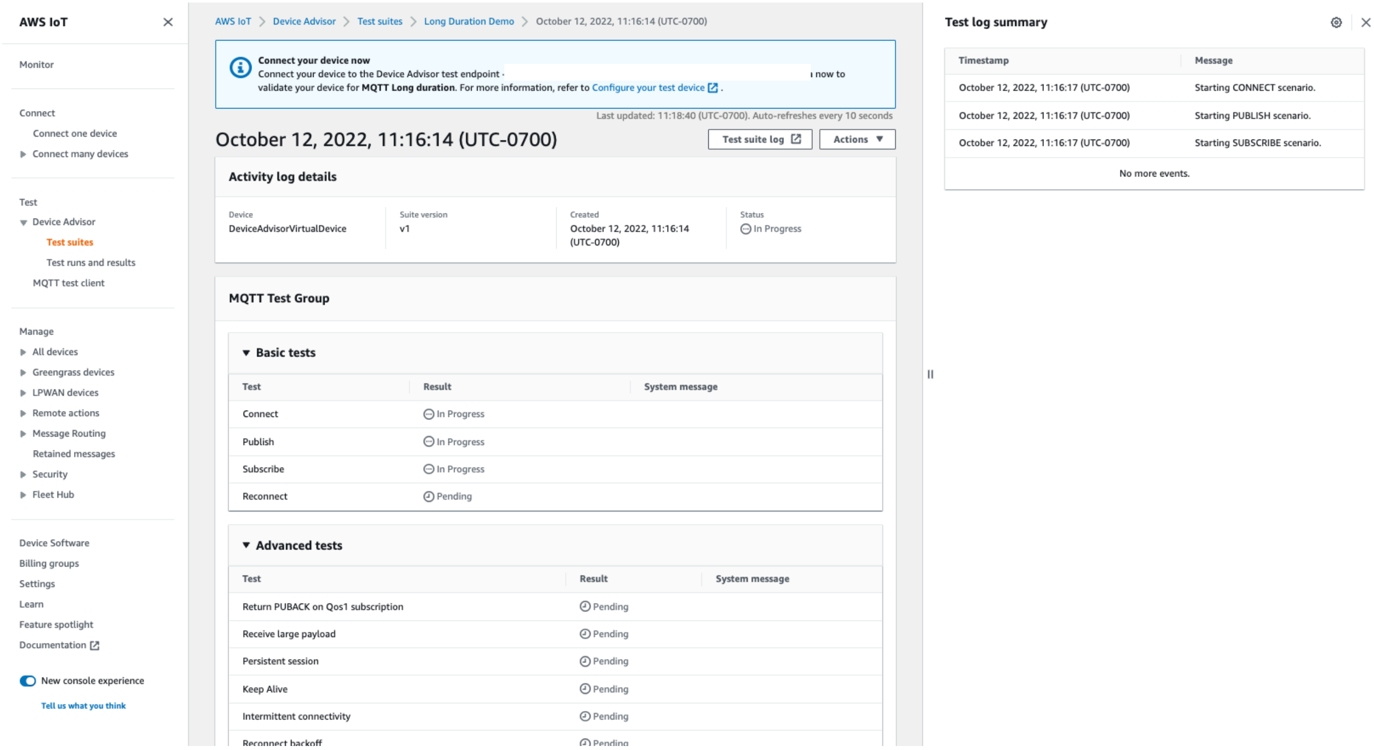
    c. Choose **Run test** to run the test.



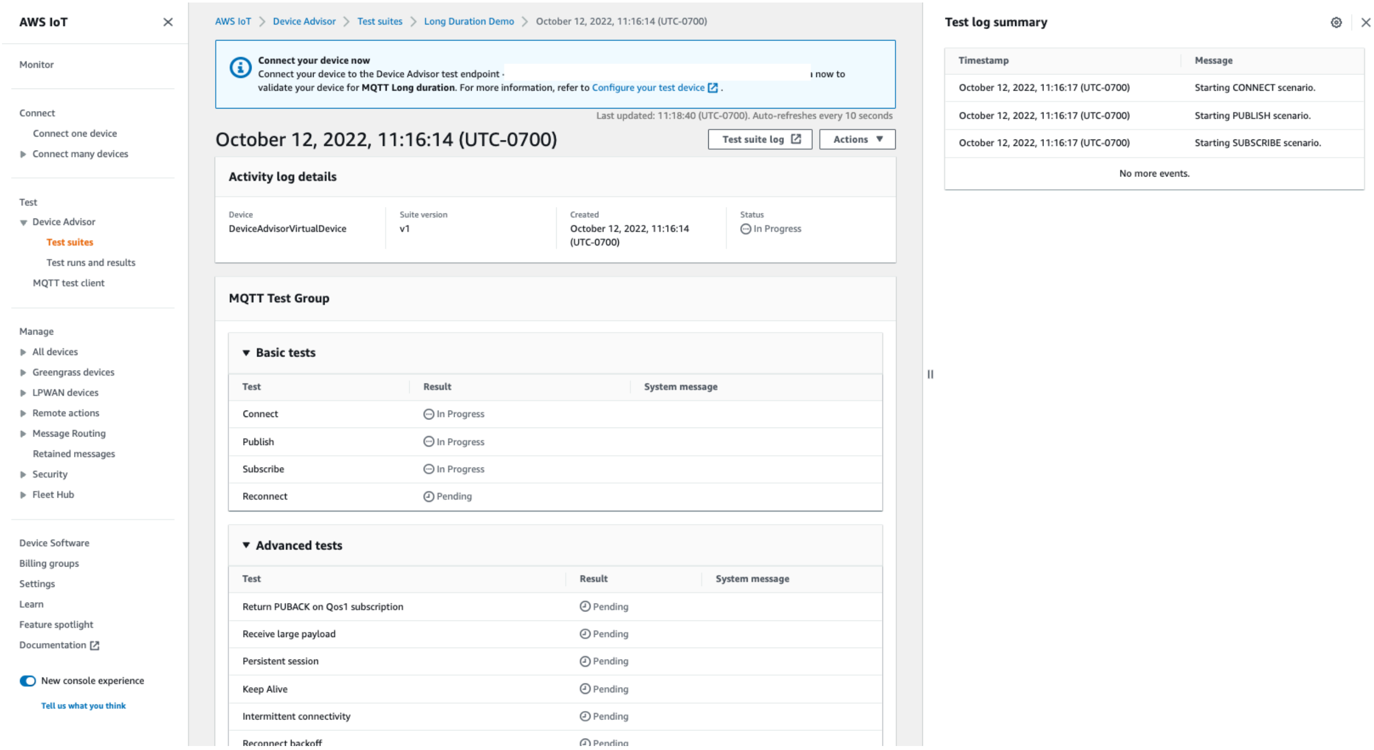
11. To view the results of the test suite run, select **Test runs and results** in the left navigation pane. Choose the test suite that ran to view the details of the results.



12. The previous step brings up the test summary page. All the details of the test run are displayed in this page. When the console prompts to start the device connection, connect your device to the provided endpoint. The progress of the tests is seen on this page.



13. The Long duration test provides an additional **Test log summary** on the side panel which displays all the important events occurring between the device and the broker in near real time. To view more in-depth detailed logs, click on **Test case log**.



Device Advisor VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and the AWS IoT Core Device Advisor test endpoint (data plane) by creating an *interface VPC endpoint*. You can use this endpoint to validate AWS IoT devices for reliable and secure connectivity with AWS IoT Core before deploying devices to production. Device Advisor's pre-built tests helps you validate your device software against best practices for usage of TLS, MQTT, Device Shadow, and AWS IoT Jobs.

AWS PrivateLink powers the interface endpoints used with your IoT devices. This service helps you access the AWS IoT Core Device Advisor test endpoint privately without an internet gateway, NAT device, VPN connection, or Direct Connect connection. Instances in your VPC that send TCP and MQTT packets don't need public IP addresses to communicate with AWS IoT Core Device Advisor test endpoints. Traffic between your VPC and AWS IoT Core Device Advisor doesn't leave AWS Cloud. Any TLS and MQTT communication between IoT devices and Device Advisor test cases stay within the resources in your AWS account.

Each interface endpoint is represented by one or more elastic network interfaces in your subnets.

To learn more about using interface VPC endpoints, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.

Considerations for AWS IoT Core Device Advisor VPC endpoints

Review the interface endpoint properties and limitations in the *Amazon VPC User Guide* before setting up interface VPC endpoints. Consider the following before you continue:

- AWS IoT Core Device Advisor currently supports making calls to Device Advisor test endpoint (data plane) from your VPC. A message broker uses data plane communications to send and receive data. It does this with the help of TLS and MQTT packets. VPC endpoints for AWS IoT Core Device Advisor connect your AWS IoT device to Device Advisor test endpoints. Control plane API actions aren't used by this VPC endpoint. To create or run a test suite or other control plane APIs, use the console, an AWS SDK, or AWS Command Line Interface over the public internet.
- The following AWS Regions support VPC endpoints for AWS IoT Core Device Advisor:
  - US East (N. Virginia)
  - US West (Oregon)
  - Asia Pacific (Tokyo)
  - Europe (Ireland)

- Device Advisor supports MQTT with X.509 client certificates and RSA server certificates.
- VPC endpoint policies aren't supported at this time.
- Check VPC endpoint prerequisites for instructions on how to create resources that connect VPC endpoints. You must create a VPC and private subnets to use AWS IoT Core Device Advisor VPC endpoints.
- There are quotas on your AWS PrivateLink resources. For more information, see AWS PrivateLink quotas.
- VPC endpoints support only IPv4 traffic.

Create an interface VPC endpoint for AWS IoT Core Device Advisor

To get started with VPC endpoints, create an interface VPC endpoint. Next, select AWS IoT Core Device Advisor as the AWS service. If you are using the AWS CLI, call describe-vpc-endpoint-services to confirm that AWS IoT Core Device Advisor is present in an Availability Zone in your AWS Region. Confirm that the security group attached to the endpoint allows TCP protocol communication for MQTT and TLS traffic. For example, in the US East (N. Virginia) Region, use the following command:

```
aws ec2 describe-vpc-endpoint-services --service-name com.amazonaws.us-
east-1.deviceadvisor.iot
```

You can create a VPC endpoint for AWS IoT Core using the following service name:

- com.amazonaws.*region*.deviceadvisor.iot

By default, private DNS is turned on for the endpoint. This ensures that use of the default test endpoint stays within your private subnets. To get your account or device level endpoint, use the console, AWS CLI or an AWS SDK. For example, if you run get-endpoint within a public subnet or on the public internet, you can get your endpoint and use it to connect to Device Advisor. For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

To connect MQTT clients to the VPC endpoint interfaces, the AWS PrivateLink service creates DNS records in a private hosted zone attached to your VPC. These DNS records direct the AWS IoT device's requests to the VPC endpoint.

Controlling access to AWS IoT Core Device Advisor over VPC endpoints

You can restrict device access to AWS IoT Core Device Advisor and allow access only through VPC endpoints by using VPC condition context keys. AWS IoT Core supports the following VPC related context keys:

- SourceVpc
- SourceVpce
- VPCSourceIp

Note

AWS IoT Core Device Advisor doesn't support VPC endpoint policies at this time.

The following policy grants permission to connect to AWS IoT Core Device Advisor using a client ID that matches the thing name. It also publishes to any topic prefixed by the thing name. The policy is conditional on the device connecting to a VPC endpoint with a particular VPC endpoint ID. This policy denies connection attempts to your public AWS IoT Core Device Advisor test endpoint.

```
{
"Version":"2012-10-17",
    "Statement": [
        {
"Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:client/
${iot:Connection.Thing.ThingName}"
            ],
            "Condition": {
"StringEquals": {
"aws:SourceVpce": "vpce-1a2b3c4d"
            }
        }

        },
```

```
        {
"Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/
${iot:Connection.Thing.ThingName}/*"
            ]
        }
    ]
}
```

Device Advisor test cases

Device Advisor provides prebuilt tests in six categories.

- TLS
- MQTT
- Shadow
- Job execution
- Permissions and policies
- Long duration tests

Device Advisor test cases to qualify for the AWS Device Qualification Program.

Your device must pass the following tests to qualify according to the AWS Device Qualification Program.

> **Note**
>
> This is a revised list of the qualification tests.

- TLS Connect ("TLS Connect")
- TLS Incorrect Subject Name Server Cert ("Incorrect Subject Common Name (CN) / Subject Alternative Name (SAN)")
- TLS Unsecure Server Cert ("Not Signed By Recognized CA")
- TLS Device Support for AWS IoT Cipher Suites ("TLS Device Support for AWS IoT recommended Cipher Suites")
- TLS Receive Maximum Size Fragments("TLS Receive Maximum Size Fragments")
- TLS Expired Server Cert("Expired server certificate")
- TLS Large Size Server Cert("TLS large Size Server Certificate")
- MQTT Connect ("Device send CONNECT to AWS IoT Core (Happy case)")
- MQTT Subscribe ("Can Subscribe (Happy Case)")
- MQTT Publish ("QoS0 (Happy Case)")
- MQTT Connect Jitter Retries("Device connect retries with jitter backoff - No CONNACK response")

TLS

Use these tests to determine if the transport layer security protocol (TLS) between your devices and AWS IoT is secure.

> **Note**
>
> Device Advisor now supports TLS 1.3.

Happy Path

TLS Connect

Validates if the device under test can complete the TLS handshake to AWS IoT. This test doesn't validate the MQTT implementation of the client device.

Example API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. For best results, we recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_tls_connect_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  //in seconds
      },
      "test":{
         "id":"TLS_Connect",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test completed TLS handshake with AWS IoT.
- **Pass with warnings** — The device under test completed TLS handshake with AWS IoT, but there were TLS warning messages from the device or AWS IoT.
- **Fail** — The device under test failed to complete TLS handshake with AWS IoT due to handshake error.

TLS Receive Maximum Size Fragments

This test case validates that your device can receive and process TLS maximum size fragments. Your test device must subscribe to a pre-configured topic with QoS 1 to receive a large payload. You can customize the payload with the configuration `${payload}`.

Example API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. For best results, we recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"TLS Receive Maximum Size Fragments",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  //in seconds
         "PAYLOAD_FORMAT":"{"message":"${payload}"}", // A string with a placeholder
 ${payload}, or leave it empty to receive a plain string.
         "TRIGGER_TOPIC": "test_1" // A topic to which a device will subscribe, and
 to which a test case will publish a large payload.
      },
      "test":{
         "id":"TLS_Receive_Maximum_Size_Fragments",
         "version":"0.0.0"
      }
   }
]
```

Cipher Suites

TLS Device Support for AWS IoT recommended Cipher Suites

Validates that the cipher suites in the TLS Client Hello message from the device under test contains the recommended AWS IoT cipher suites. It provides more insights into cipher suites supported by the device.

Example API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_tls_support_aws_iot_cipher_suites_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
      },
      "test":{
         "id":"TLS_Support_AWS_IoT_Cipher_Suites",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test cipher suites contain at least one of the recommended AWS IoT cipher suite and don't contain any unsupported cipher suites.
- **Pass with warnings** — The device cipher suites contain at least one AWS IoT cipher suite but:
  1. It doesn't contain any of the recommended cipher suites
  2. It contains cipher suites that aren't supported by AWS IoT.

  We suggest that you verify that any unsupported cipher suites are safe.
- **Fail** — The device under test cipher suites doesn't contain any of the AWS IoT supported cipher suites.

Larger Size Server Certificate

TLS large Size Server Certificate

Validates at your device can complete the TLS handshake with AWS IoT when it receives and processes a larger size server certificate. The size of the server certificate (in bytes) used by this test is larger than what is currently used in the **TLS Connect** test case and IoT Core by 20 During this test case, AWS IoT tests your device’s buffer space for TLS If the buffer space is large enough, the TLS handshake ompletes without errors. This test esn't validate the MQTT implementation of the device. The test case ds after the TLS handshake process completes.

Example API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. For best results, we recommend a timeout value of 2 minutes. If this test case fails but the **TLS Connect** test case passes, we recommend you increase your device's buffer space limit for TLS Increasing the buffer space limit sures that your device can process a larger size server certificate in case the size increases.

```
"tests":[
   {
      "name":"my_tls_large_size_server_cert_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
      },
      "test":{
         "id":"TLS_Large_Size_Server_Cert",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test completed the TLS handshake with AWS IoT.
- **Pass with warnings** — The device under test completed the TLS handshake with AWS IoT, but there are TLS warning messages either from the device or AWS IoT.
- **Fail** — The device under test failed to complete the TLS handshake with AWS IoT because of an error during the handshake process.

TLS Unsecure Server Cert

Not Signed By Recognized CA

Validates that the device under test closes the connection if it's presented with a server certificate without a valid signature from the ATS CA. A device should only connect to an endpoint that presents a valid certificate.

Example API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_tls_unsecure_server_cert_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  //in seconds
      },
      "test":{
         "id":"TLS_Unsecure_Server_Cert",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test closed the connection.
- **Fail** — The device under test completed TLS handshake with AWS IoT.

TLS Incorrect Subject Name Server Cert / Incorrect Subject Common Name (CN) / Subject Alternative Name (SAN)

Validates that the device under test closes the connection if it's presented with a server certificate for a domain name that is different than the one requested.

Example API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_tls_incorrect_subject_name_cert_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",   // in seconds
      },
      "test":{
         "id":"TLS_Incorrect_Subject_Name_Server_Cert",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test closed the connection.
- **Fail** — The device under test completed the TLS handshake with AWS IoT.

TLS Expired Server Certificate

Expired server certificate

Validates that the device under test closes the connection if it's presented with an expired server certificate.

Example API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_tls_expired_cert_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",  //in seconds
      },
      "test":{
         "id":"TLS_Expired_Server_Cert",
         "version":"0.0.0"
      }
   }
]
```

Example Test case outputs:

- **Pass** — The device under test refuses to complete the TLS handshake with AWS IoT. The device sends a TLS alert message before it closes the connection.
- **Pass with warnings** — The device under test refuses to complete the TLS handshake with AWS IoT. However, it doesn't send a TLS alert message before it closes the connection.
- **Fail** — The device under test completes the TLS handshake with AWS IoT.

MQTT

CONNECT, DISCONNECT, and RECONNECT

"Device send CONNECT to AWS IoT Core (Happy case)"

Validates that the device under test sends a CONNECT request.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
```

```
    {
       "name":"my_mqtt_connect_test",
       "configuration": {
          // optional:
          "EXECUTION_TIMEOUT":"300",   // in seconds
       },
       "test":{
          "id":"MQTT_Connect",
          "version":"0.0.0"
       }
    }
]
```

"Device can return PUBACK to an arbitrary topic for QoS1"

This test case will check if the device (client) can return a PUBACK message if it received a publish message from the broker after subscribing to a topic with QoS1.

The payload content and the payload size are configurable for this test case. If the payload size is configured, Device Advisor will overwrite the value for the payload content, and send a predefined payload to the device with the desired size. The payload size is a value between 0 to 128 and cannot exceed 128 KB. AWS IoT Core rejects publish and connect requests larger than 128 KB, as seen in the AWS IoT Core message broker and protocol limits and quotas page.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes. `PAYLOAD_SIZE` can be configured to a value between 0 and 128 kilobytes. Defining a payload size overrides the payload content as Device Advisor will be sending a pre-defined payload with the given size back to the device.

```
"tests":[
{
        "name":"my_mqtt_client_puback_qos1",
        "configuration": {
            // optional:"TRIGGER_TOPIC": "myTopic",
            "EXECUTION_TIMEOUT":"300", // in seconds
            "PAYLOAD_FOR_PUBLISH_VALIDATION":"custom payload",
```

```
            "PAYLOAD_SIZE":"100" // in kilobytes
        },
        "test": {
            "id": "MQTT_Client_Puback_QoS1",
            "version": "0.0.0"
        }
    }
]
```

"Device connect retries with jitter backoff - No CONNACK response"

Validates that the device under test uses the proper jitter backoff when reconnecting with the broker for at least five times. The broker logs the timestamp of the device under test's CONNECT request, performs packet validation, pauses without sending a CONNACK to the device under test, and waits for the device under test to resend the request. The sixth connection attempt is allowed to pass through and CONNACK is allowed to flow back to the device under test.

The preceding process is performed again. In total, this test case requires the device to connect at least 12 times in total. The collected timestamps are used to validate that jitter backoff is used by the device under test. If the device under test has a strictly exponential backoff delay, this test case will pass with warnings.

We recommend implementation of the [Exponential Backoff And Jitter](https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/) mechanism on the device under test to pass this test case.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```
"tests":[
   {
      "name":"my_mqtt_jitter_backoff_retries_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300",    // in seconds
```

```
      },
      "test":{
         "id":"MQTT_Connect_Jitter_Backoff_Retries",
         "version":"0.0.0"
      }
   }
]
```

"Device connect retries with exponential backoff - No CONNACK response"

Validates that the device under test uses the proper exponential backoff when reconnecting with the broker for at least five times. The broker logs the timestamp of the device under test's CONNECT request, performs packet validation, pauses without sending a CONNACK to the client device, and waits for the device under test to resend the request. The collected timestamps are used to validate that an exponential backoff is used by the device under test.

We recommend implementation of the [Exponential Backoff And Jitter](#) mechanism on the device under test to pass this test case.

API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```
"tests":[
   {
      "name":"my_mqtt_exponential_backoff_retries_test",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"600",  // in seconds
      },
      "test":{
         "id":"MQTT_Connect_Exponential_Backoff_Retries",
         "version":"0.0.0"
      }
   }
]
```

"Device re-connect with jitter backoff - After server disconnect"

Validates if a device under test uses necessary jitter and backoff while reconnecting after it's been disconnected from the server. Device Advisor disconnects the device from the server for at least five times and observes the device's behavior for MQTT reconnection. Device Advisor logs the timestamp of the CONNECT request for the device under test, performs packet validation, pauses without sending a CONNACK to the client device, and waits for the device under test to resend the request. The collected timestamps are used to validate that the device under test uses jitter and backoff while reconnecting. If the device under test has a strictly exponential backoff or doesn't implement a proper jitter backoff mechanism, this test case will pass with warnings. If the device under test has implemented either a linear backoff or a constant backoff mechanism, the test will fail.

To pass this test case, we recommend implementing the Exponential Backoff And Jitter mechanism on the device under test.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 4 minutes.
> The number of reconnection attempts to validate for backoff can be changed by specifying the `RECONNECTION_ATTEMPTS`. The number must be between 5 and 10. The default value is 5.

```
"tests":[
   {
      "name":"my_mqtt_reconnect_backoff_retries_on_server_disconnect",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
         "RECONNECTION_ATTEMPTS": 5
      },
      "test":{
         "id":"MQTT_Reconnect_Backoff_Retries_On_Server_Disconnect",
         "version":"0.0.0"
      }
   }
```

```
]
```

"Device re-connect with jitter backoff - On unstable connection"

Validates if a device under test uses necessary jitter and backoff while reconnecting on an unstable connection. Device Advisor disconnects the device from the server after five successful connections, and observes the device's behavior for MQTT reconnection. Device Advisor logs the timestamp of the CONNECT request for the device under test, performs packet validation, sends back CONNACK, disconnects, log the timestamp of the disconnection, and waits for the device under test to resend the request. The collected timestamps are used to validate that the device under test uses jitter and backoff while reconnecting after successful but unstable connections. If the device under test has a strictly exponential backoff or doesn't implement a proper jitter backoff mechanism, this test case will pass with warnings. If the device under test has implemented either a linear backoff or a constant backoff mechanism, the test will fail.

To pass this test case, we recommend implementing the [Exponential Backoff And Jitter](#) mechanism on the device under test.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 4 minutes.
> The number of reconnection attempts to validate for backoff can be changed by specifying the `RECONNECTION_ATTEMPTS`. The number must be between 5 and 10. The default value is 5.

```
"tests":[
   {
      "name":"my_mqtt_reconnect_backoff_retries_on_unstable_connection",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
         "RECONNECTION_ATTEMPTS": 5
      },
      "test":{
         "id":"MQTT_Reconnect_Backoff_Retries_On_Unstable_Connection",
```

```
            "version":"0.0.0"
        }
    }
]
```

Publish

"QoS0 (Happy Case)"

Validates that the device under test publishes a message with QoS0 or QoS1. You can also validate the topic of the message and payload by specifying the topic value and payload in the test settings.

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_mqtt_publish_test",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
         "TOPIC_FOR_PUBLISH_VALIDATION": "my_TOPIC_FOR_PUBLISH_VALIDATION",
         "PAYLOAD_FOR_PUBLISH_VALIDATION": "my_PAYLOAD_FOR_PUBLISH_VALIDATION",
      },
      "test":{
         "id":"MQTT_Publish",
         "version":"0.0.0"
      }
   }
]
```

"QoS1 publish retry - No PUBACK"

Validates that the device under test republishes a message sent with QoS1, if the broker doesn't send PUBACK. You can also validate the topic of the message by specifying this topic in the test settings. The client device must not disconnect before republishing the message. This test also validates that the republished message has the same packet identifier as the original. During the test execution, if the device loses connection and reconnects, the test case will reset without failing and the device has to perform the test case steps again.

API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. It is recommended for at least 4 minutes.

```
"tests":[
   {
      "name":"my_mqtt_publish_retry_test",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
         "TOPIC_FOR_PUBLISH_VALIDATION": "my_TOPIC_FOR_PUBLISH_VALIDATION",
         "PAYLOAD_FOR_PUBLISH_VALIDATION": "my_PAYLOAD_FOR_PUBLISH_VALIDATION",
      },
      "test":{
         "id":"MQTT_Publish_Retry_No_Puback",
         "version":"0.0.0"
      }
   }
]
```

"Publish Retained messages"

Validates that the device under test publishes a message with `retainFlag` set to true. You can validate the topic and payload of the message by setting the topic value and payload in the test settings. If the `retainFlag` sent within the PUBLISH packet is not set to true, the test case will fail.

API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes. To run this test case, add the iot:RetainPublish action in your device role.

```
"tests":[
   {
      "name":"my_mqtt_publish_retained_messages_test",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds

 "TOPIC_FOR_PUBLISH_RETAINED_VALIDATION": "my_TOPIC_FOR_PUBLISH_RETAINED_VALIDATION",

 "PAYLOAD_FOR_PUBLISH_RETAINED_VALIDATION": "my_PAYLOAD_FOR_PUBLISH_RETAINED_VALIDATION",
      },
      "test":{
         "id":"MQTT_Publish_Retained_Messages",
         "version":"0.0.0"
      }
   }
]
```

"Publish with User Property"

Validates that the device under test publishes a message with the correct user property. You can validate the user property by setting the name-value pair in the test settings. If the user property is not provided or doesn't match, the test case fails.

API test case definition:

> **Note**
>
> This is a MQTT5 only test case.
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_mqtt_user_property_test",
      "test":{
        "USER_PROPERTIES": [
            {"name": "name1", "value":"value1"},
            {"name": "name2", "value":"value2"}
        ],
        "EXECUTION_TIMEOUT":"300",  // in seconds
      },
      "test":{
         "id":"MQTT_Publish_User_Property",
         "version":"0.0.0"
      }
   }
]
```

Subscribe

"Can Subscribe (Happy Case)"

Validates that the device under test subscribes to MQTT topics. You can also validate the topic that the device under test subscribes to by specifying this topic in the test settings.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_mqtt_subscribe_test",
      "configuration":{
         // optional:
         "EXECUTION_TIMEOUT":"300",  // in seconds
         "TOPIC_LIST_FOR_SUBSCRIPTION_VALIDATION":
["my_TOPIC_FOR_PUBLISH_VALIDATION_a","my_TOPIC_FOR_PUBLISH_VALIDATION_b"]
```

```
      },
      "test":{
         "id":"MQTT_Subscribe",
         "version":"0.0.0"
      }
   }
]
```

"Subscribe Retry - No SUBACK"

Validates that the device under test retries a failed subscription to MQTT topics. The server then waits and doesn't send a SUBACK. If the client device doesn't retry the subscription, the test fails. The client device must retry the failed subscription with the same packet Id. You can also validate the topic that the device under test subscribes to by specifying this topic in the test settings. During the test execution, if the device loses connection and reconnects, the test case will reset without failing and the device has to perform the test case steps again.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 4 minutes.

```
"tests":[
   {
      "name":"my_mqtt_subscribe_retry_test",
      "configuration":{
         "EXECUTION_TIMEOUT":"300",  // in seconds
         // optional:
         "TOPIC_LIST_FOR_SUBSCRIPTION_VALIDATION":
["myTOPIC_FOR_PUBLISH_VALIDATION_a","my_TOPIC_FOR_PUBLISH_VALIDATION_b"]
      },
      "test":{
         "id":"MQTT_Subscribe_Retry_No_Suback",
         "version":"0.0.0"
      }
   }
]
```

Keep-Alive

"Mqtt No Ack PingResp"

This test case validates if the device under test disconnects when it doesn't receive a ping response. As part of this test case, Device Advisor blocks responses sent from AWS IoT Core for publish, subscribe, and ping requests. It also validates if the device under test disconnects the MQTT connection.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout greater than 1.5 times the `keepAliveTime` value.
> The maximum `keepAliveTime` must be no greater than 230 seconds for this test.

```
"tests":[
    {
       "name":"Mqtt No Ack PingResp",
       "configuration":
          //optional:
          "EXECUTION_TIMEOUT":"306",   // in seconds
       },
       "test":{
          "id":"MQTT_No_Ack_PingResp",
          "version":"0.0.0"
       }
    }
]
```

Persistent Session

"Persistent Session (Happy Case)"

This test case validates the device behavior when disconnected from a persistent session. The test case checks if the device can reconnect, resume the subscriptions to its trigger topics without explicitly re-subscribing, receive the stored messages in the topics, and work as expected during a persistent session. When this test case passes, it indicates that the client device is able to maintain a persistent session with the AWS IoT Core broker in an expected manner. For more information on AWS IoT Persistent Sessions, see [Using MQTT persistent sessions](#).

In this test case, the client device is expected to CONNECT with the AWS IoT Core with a clean session flag set to false, and then subscribe to a trigger topic. After a successful subscription, the device will be disconnected by AWS IoT Core Device Advisor. While the device is in a disconnected state, a QoS 1 message payload will be stored in that topic. Device Advisor will then allow the client device to re-connect with the test endpoint. At this point, since there is a persistent session, the client device is expected to resume its topic subscriptions without sending any additional SUBSCRIBE packets and receive the QoS 1 message from the broker. After re-connecting, if the client device re-subscribes to its trigger topic again by sending an additional SUBSCRIBE packet and/or if the client fails to receive the stored message from the trigger topic, the test case will fail.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of at least 4 minutes. In the first connection, client device needs to explicitly subscribe to a `TRIGGER_TOPIC` which was not subscribed before. To pass the test case, client device must successfully subscribe to `TRIGGER_TOPIC` with a QoS 1. After re-connecting, the client device is expected to understand that there is an active persistent session; so it should accept the stored message sent by the trigger topic and return PUBACK for that specific message.

```
"tests":[
   {
      "name":"my_mqtt_persistent_session_happy_case",
      "configuration":{
         //required:
         "TRIGGER_TOPIC": "myTrigger/topic",
         // optional:
         // if Payload not provided, a string will be stored in the trigger topic to
 be sent back to the client device
         "PAYLOAD": "The message which should be received from AWS IoT Broker after
 re-connecting to a persistent session from the specified trigger topic.",
```

```
            "EXECUTION_TIMEOUT":"300" // in seconds
        },
        "test":{
            "id":"MQTT_Persistent_Session_Happy_Case",
            "version":"0.0.0"
        }
    }
]
```

"Persistent Session - Session Expiry"

This test case helps to validate device behavior when a disconnected device reconnects to an expired persistent session. After the session expires, we expect the device to resubscribe to the topics previously subscribed to by explicitly sending a new SUBSCRIBE packet.

During the first connection, we expect the test device to CONNECT with the AWS IoT broker, as its `CleanSession` flag is set to false to initiate a persistent session. The device should then subscribe to a trigger topic. Then the device is disconnected by AWS IoT Core Device Advisor, after a successful subscription and initiation of a persistent session. After the disconnection, AWS IoT Core Device Advisor allows the test device to re-connect back with the test endpoint. At this point, when the test device sends another CONNECT packet, AWS IoT Core Device Advisor sends back a CONNACK packet that indicates that the persistent session is expired. The test device needs to interpret this packet properly, and it is expected to re-subscribe to the same trigger topic again as the persistent session is terminated. If the test device does not re-subscribe to its topic trigger again, the test case fails. For the test to pass, the device needs to understand that the persistent session is over, and send back a new SUBSCRIBE packet for the same trigger topic in the second connection.

If this test case passes for a test device, it indicates that the device is able to handle re-connection on expiry of persistent session in an expected way.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of at least 4 minutes. The test device needs to explicitly subscribe to a `TRIGGER_TOPIC`, to which it was not subscribed before. To pass the test case, the test device must send a CONNECT packet with `CleanSession` flag set to false, and successfully subscribe to a trigger topic with a QoS 1. After a successful connection, AWS IoT Core Device

Advisor disconnects the device. After the disconnection, AWS IoT Core Device Advisor allows the device to re-connect back, and the device is expected to re-subscribe to the same TRIGGER_TOPIC since AWS IoT Core Device Advisor would have terminated the persistent session.

```
"tests":[
   {
      "name":"my_expired_persistent_session_test",
      "configuration":{
         //required:
         "TRIGGER_TOPIC": "myTrigger/topic",
         // optional:
         "EXECUTION_TIMEOUT":"300" // in seconds
      },
      "test":{
         "id":"MQTT_Expired_Persistent_Session",
         "version":"0.0.0"
      }
   }
]
```

Shadow

Use these tests to verify your devices under test use AWS IoT Device Shadow service correctly. See AWS IoT Device Shadow service for more information. If these test cases are configured in your test suite, then providing a thing is required when starting the suite run.

MQTT over WebSocket is not supported at this time.

Publish

"Device publishes state after it connects (Happy case)"

Validates if a device can publish its state after it connects to AWS IoT Core

API test case definition:

> **Note**
>
> EXECUTION_TIMEOUT has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_shadow_publish_reported_state",
      "configuration": {
         // optional:
         "EXECUTION_TIMEOUT":"300", // in seconds
         "SHADOW_NAME": "SHADOW_NAME",
         "REPORTED_STATE": {
            "STATE_ATTRIBUTE": "STATE_VALUE"
         }
      },
      "test":{
         "id":"Shadow_Publish_Reported_State",
         "version":"0.0.0"
      }
   }
]
```

The `REPORTED_STATE` can be provided for additional validation on your device's exact shadow state, after it connects. By default, this test case validates your device publishing state.

If *SHADOW_NAME* is not provided, the test case looks for messages published to topic prefixes of the Unnamed (classic) shadow type by default. Provide a shadow name if your device uses the named shadow type. See Using shadows in devices for more information.

Update

"Device updates reported state to desired state (Happy case)"

Validates if your device reads all update messages received and synchronizes the device's state to match the desired state properties. Your device should publish its latest reported state after synchronizing. If your device already has an existing shadow before running the test, make sure the desired state configured for the test case and the existing reported state do not already match. You can identify Shadow update messages sent by

Device Advisor by looking at the **ClientToken** field in the Shadow document as it will be `DeviceAdvisorShadowTestCaseSetup`.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 2 minutes.

```
"tests":[
   {
      "name":"my_shadow_update_reported_state",
      "configuration": {
         "DESIRED_STATE": {
            "STATE_ATTRIBUTE": "STATE_VALUE"
         },
         // optional:
         "EXECUTION_TIMEOUT":"300", // in seconds
         "SHADOW_NAME": "SHADOW_NAME"
      },
      "test":{
         "id":"Shadow_Update_Reported_State",
         "version":"0.0.0"
      }
   }
]
```

The `DESIRED_STATE` should have at least one attribute and associated value.

If `SHADOW_NAME` is not provided, then the test case looks for messages published to topic prefixes of the Unnamed (classic) shadow type by default. Provide a shadow name if your device uses the named shadow type. See Using shadows in devices for more information.

Job Execution

"Device can complete a job execution"

This test case helps you validate if your device is able to receive updates using AWS IoT Jobs, and publish the status of successful updates. For more information on AWS IoT Jobs, see [Jobs].

To successfully run this test case, there are two reserved AWS topics that you need to grant your [Device Role] . To subscribe to job activity related messages, use the **notify** and **notify-next** topics. Your device role must grant PUBLISH action for the following topics:

- $aws/things/**thingName**/jobs/**jobId**/get
- $aws/things/**thingName**/jobs/**jobId**/update

It is recommended to grant SUBSCRIBE and RECEIVE actions for the following topics:

- $aws/things/**thingName**/jobs/get/accepted
- $aws/things/**thingName**/jobs/**jobId**/get/rejected
- $aws/things/**thingName**/jobs/**jobId**/update/accepted
- $aws/things/**thingName**/jobs/**jobId**/update/rejected

It is recommended to grant SUBSCRIBE action for the following topic:

- $aws/things/**thingName**/jobs/notify-next

For more information about these reserved topics, see reserved topics for [AWS IoT Jobs].

MQTT over WebSocket is not supported at this time.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 5 minutes. We recommend a timeout value of 3 minutes. Depending on the AWS IoT Job document or source provided, adjust the timeout value (for example, if a job will take a long time to run, define a longer timeout value for the test case). To run the test, either a valid AWS IoT Job document or an already existing job ID is required. An AWS IoT Job document can be provided as a JSON document or an S3 link. If a job document is provided, providing a job ID is optional. If a job ID is provided, Device Advisor will use that ID while creating the AWS IoT Job on your behalf. If the job document is not provided, you can provide an existing ID that is

in the same region as you are running the test case. In this case, Device Advisor will use that AWS IoT Job while running the test case.

```
"tests": [
   {
      "name":"my_job_execution",
      "configuration": {
         // optional:
         // Test case will create a job task by using either JOB_DOCUMENT or
 JOB_DOCUMENT_SOURCE.
         // If you manage the job task on your own, leave it empty and provide the
 JOB_JOBID (self-managed job task).
         // JOB_DOCUMENT is a JSON formatted string
         "JOB_DOCUMENT": "{
            \"operation\":\"reboot\",
            \"files\" : {
               \"fileName\" : \"install.py\",
               \"url\" : \"${aws:iot:s3-presigned-url:https://s3.amazonaws.com/
bucket-name/key}\"
            }
         }",
         // JOB_DOCUMENT_SOURCE is an S3 link to the job document. It will be used
 only if JOB_DOCUMENT is not provided.
         "JOB_DOCUMENT_SOURCE": "https://s3.amazonaws.com/bucket-name/key",
         // JOB_JOBID is mandatory, only if neither document nor document source is
 provided. (Test case needs to know the self-managed job task id).
         "JOB_JOBID": "String",
         // JOB_PRESIGN_ROLE_ARN is used for the presign Url, which will replace the
 placeholder in the JOB_DOCUMENT field
         "JOB_PRESIGN_ROLE_ARN": "String",
         // Presigned Url expiration time. It must be between 60 and 3600 seconds,
 with the default value being 3600.
         "JOB_PRESIGN_EXPIRES_IN_SEC": "Long"
         "EXECUTION_TIMEOUT": "300", // in seconds
      },
      "test": {
         "id": "Job_Execution",
         "version": "0.0.0"
      }
   }
]
```

For more information on creating and using job documents see job document.

Permissions and policies

You can use the following tests to determine if the policies attached to your devices' certificates follow standard best practices.

MQTT over WebSocket is not supported at this time.

"Device certificate attached policies don't contain wildcards"

Validates if the permission policies associated with a device follow best practices and do not grant the device more permissions than needed.

API test case definition:

> **Note**
>
> `EXECUTION_TIMEOUT` has a default value of 1 minute. We recommend setting a timeout of at least 30 seconds.

```
"tests":[
   {
        "name":"my_security_device_policies",
        "configuration": {
            // optional:
            "EXECUTION_TIMEOUT":"60"    // in seconds
        },
        "test": {
            "id": "Security_Device_Policies",
            "version": "0.0.0"
        }
    }
]
```

Long duration tests

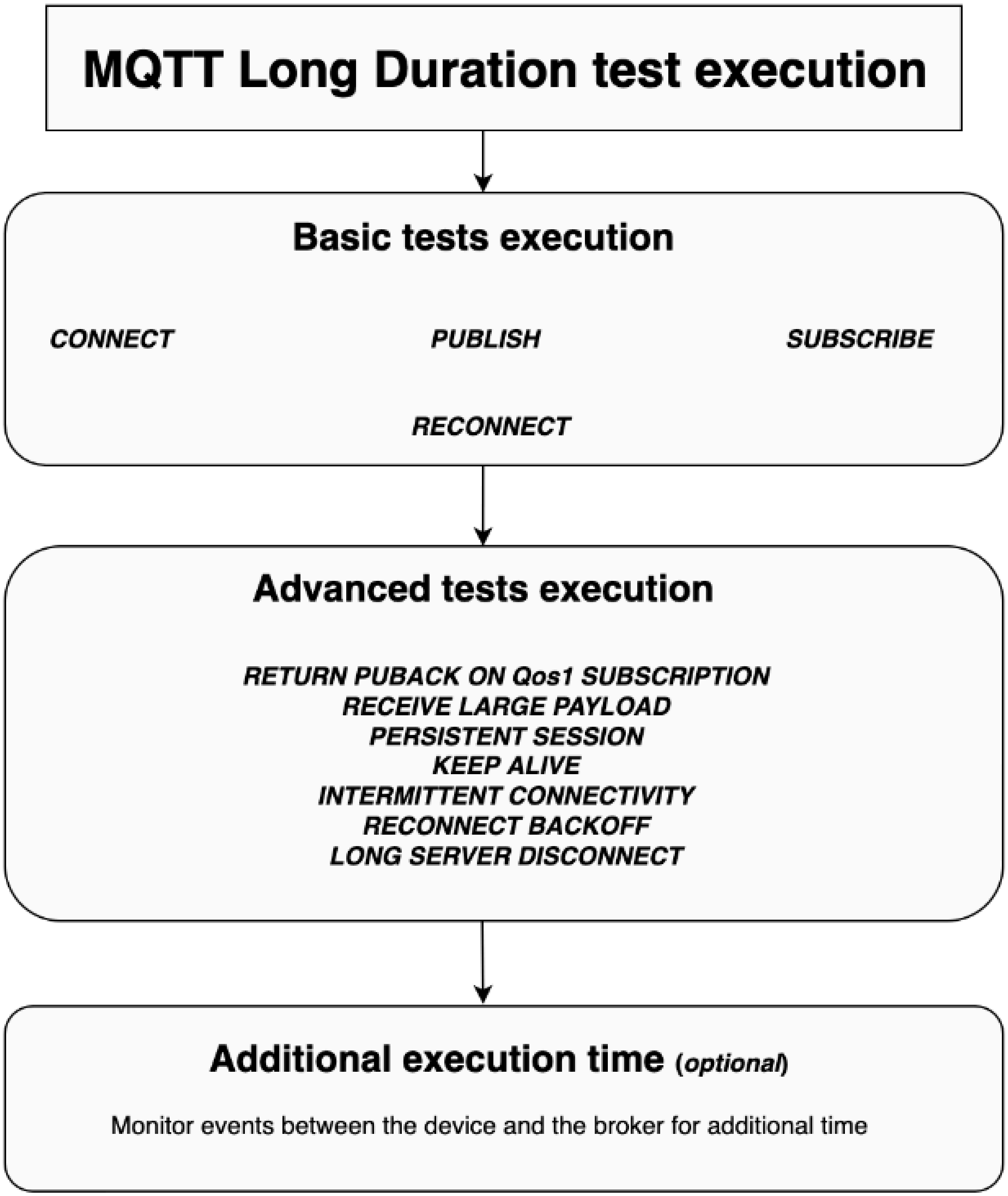
Long duration tests is a new test suite that monitors a device's behavior when it operates over longer periods of time. Compared to running individual tests that focus on specific behaviors of a device, the long duration test examines the device's behavior in a variety of real-world scenarios over the device's lifespan. Device Advisor orchestrates the tests in the most efficient possible order. The test generates results and logs, including a summary log with useful metrics about the device's performance while under test.

MQTT long duration test case

In the MQTT long duration test case, the device's behavior is initially observed in happy case scenarios such as MQTT Connect, Subscribe, Publish, and Reconnect. Then, the device is observed in multiple, complex failure scenarios such as MQTT Reconnect Backoff, Long Server Disconnect, and Intermittent Connectivity.

MQTT long duration test case execution flow

There are three phases in the execution of a MQTT long duration test case:


MQTT Long Duration test execution
Basic tests execution
CONNECT
PUBLISH
SUBSCRIBE
RECONNECT
Advanced tests execution
RETURN PUBACK ON Qos1 SUBSCRIPTION
RECEIVE LARGE PAYLOAD
PERSISTENT SESSION
KEEP ALIVE
INTERMITTENT CONNECTIVITY
RECONNECT BACKOFF
LONG SERVER DISCONNECT
Additional execution time (optional)
Monitor events between the device and the broker for additional time

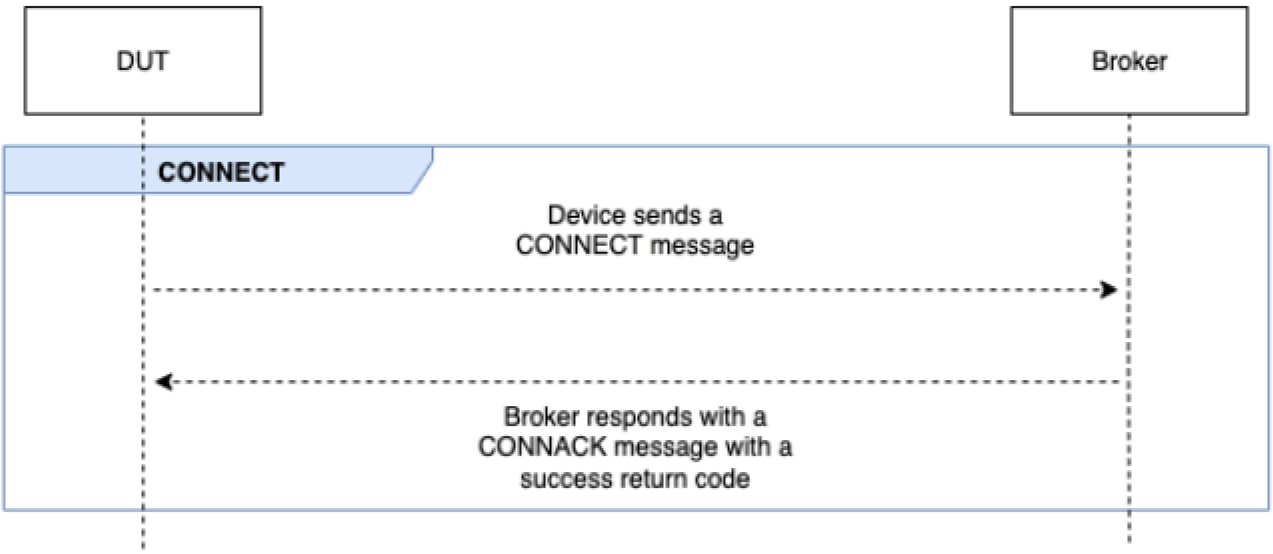
Basic tests execution

In this phase, the test case runs simple tests in parallel. The test validates if the device has the operations selected in the configuration.

The set of basic tests can include the following, based on the operations selected:

CONNECT

This scenario validates if the device is able to make a successful connection with the broker.

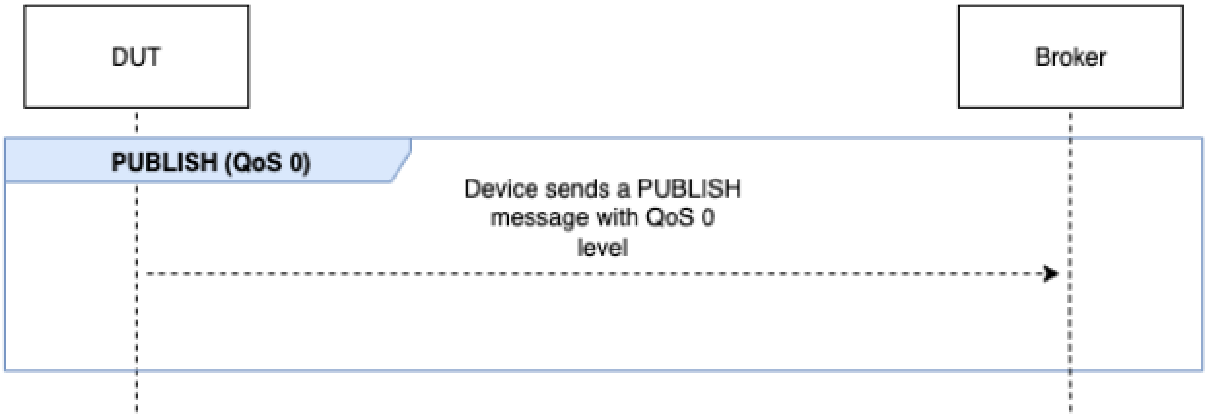


PUBLISH

This scenario validates if the device successfully publishes against the broker.

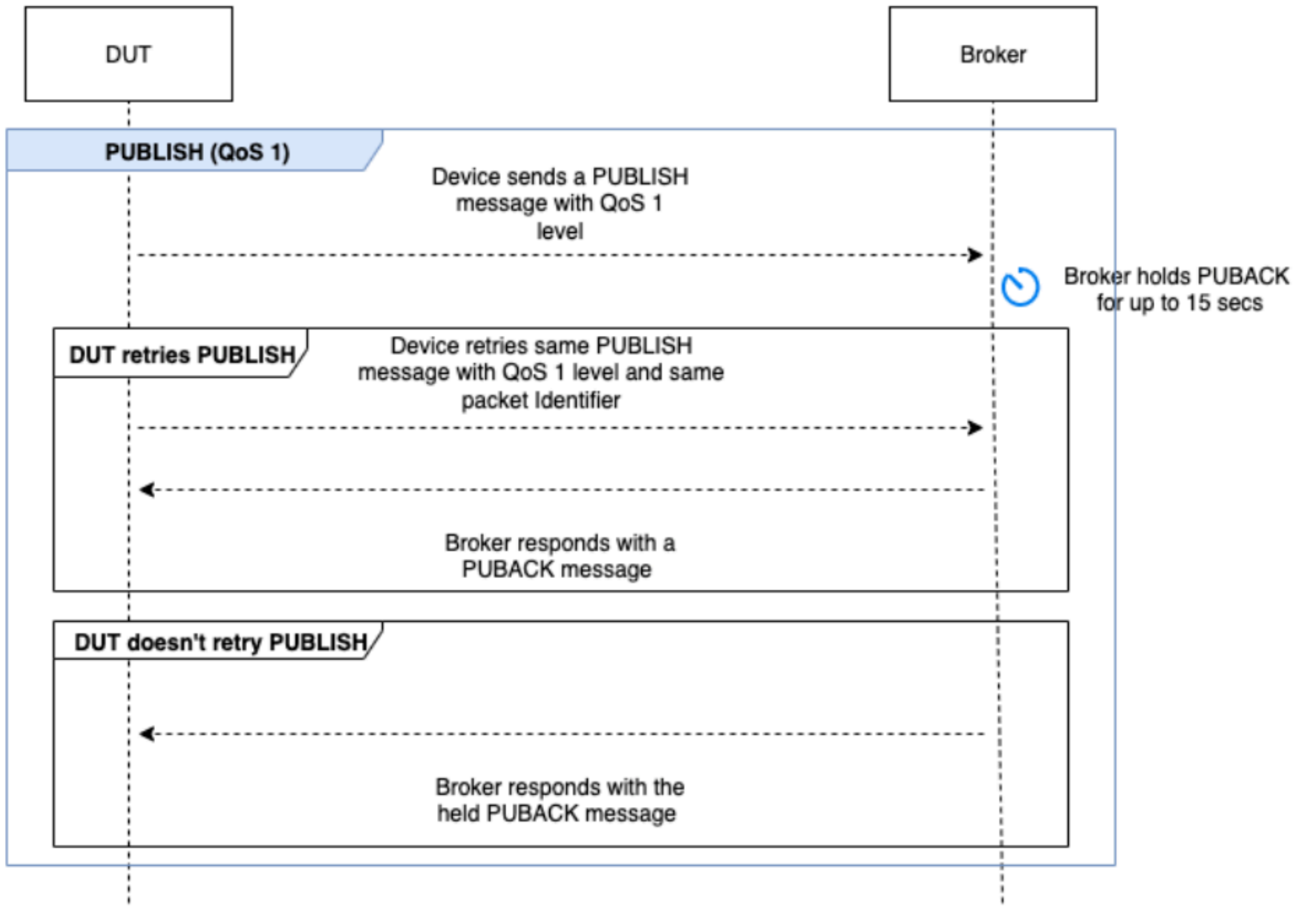
QoS 0

This test case validates if the device successfully sends a `PUBLISH` message to the broker during a publish with QoS 0. The test does not wait on the `PUBACK` message to be received by the device.



QoS 1

In this test case, the device is expected to send two PUBLISH messages to the broker with QoS 1. After the first PUBLISH message, the broker waits for up to 15 seconds before it responds. The device must retry the original PUBLISH message with the same packet identifier within the 15 second window. If it does, the broker responds with a PUBACK message and the test validates. If the device doesn't retry the PUBLISH, the original PUBACK is sent to the device and the test is marked as **Pass with warnings**, along with a system message. During the test execution, if the device loses connection and reconnects, the test scenario will reset without failing and the device has to perform the test scenario steps again.

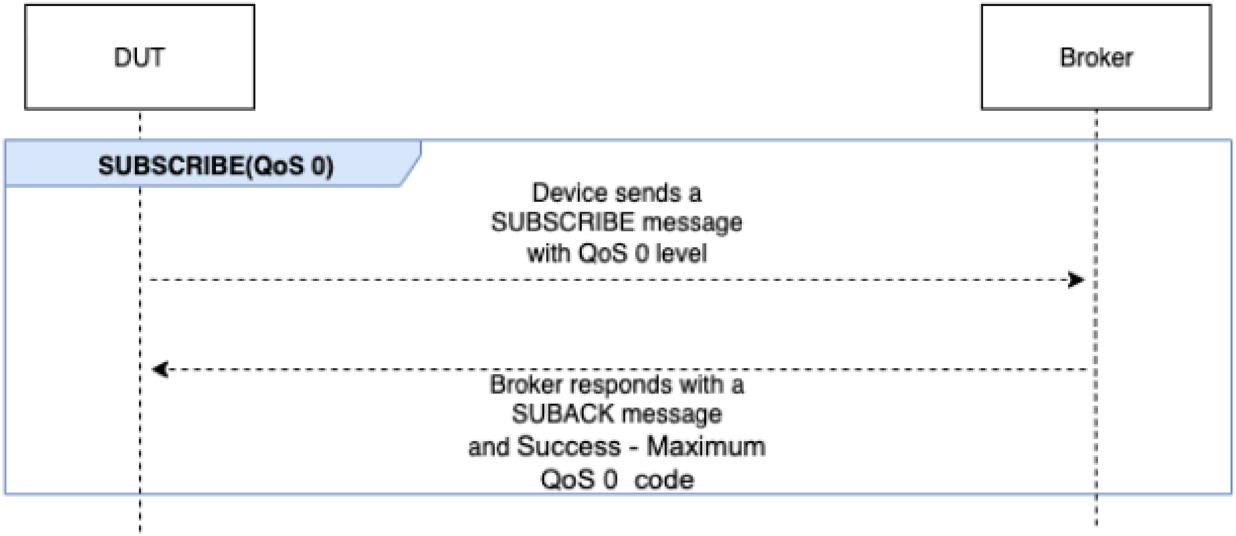


SUBSCRIBE

This scenario validates if the device successfully subscribes against the broker.

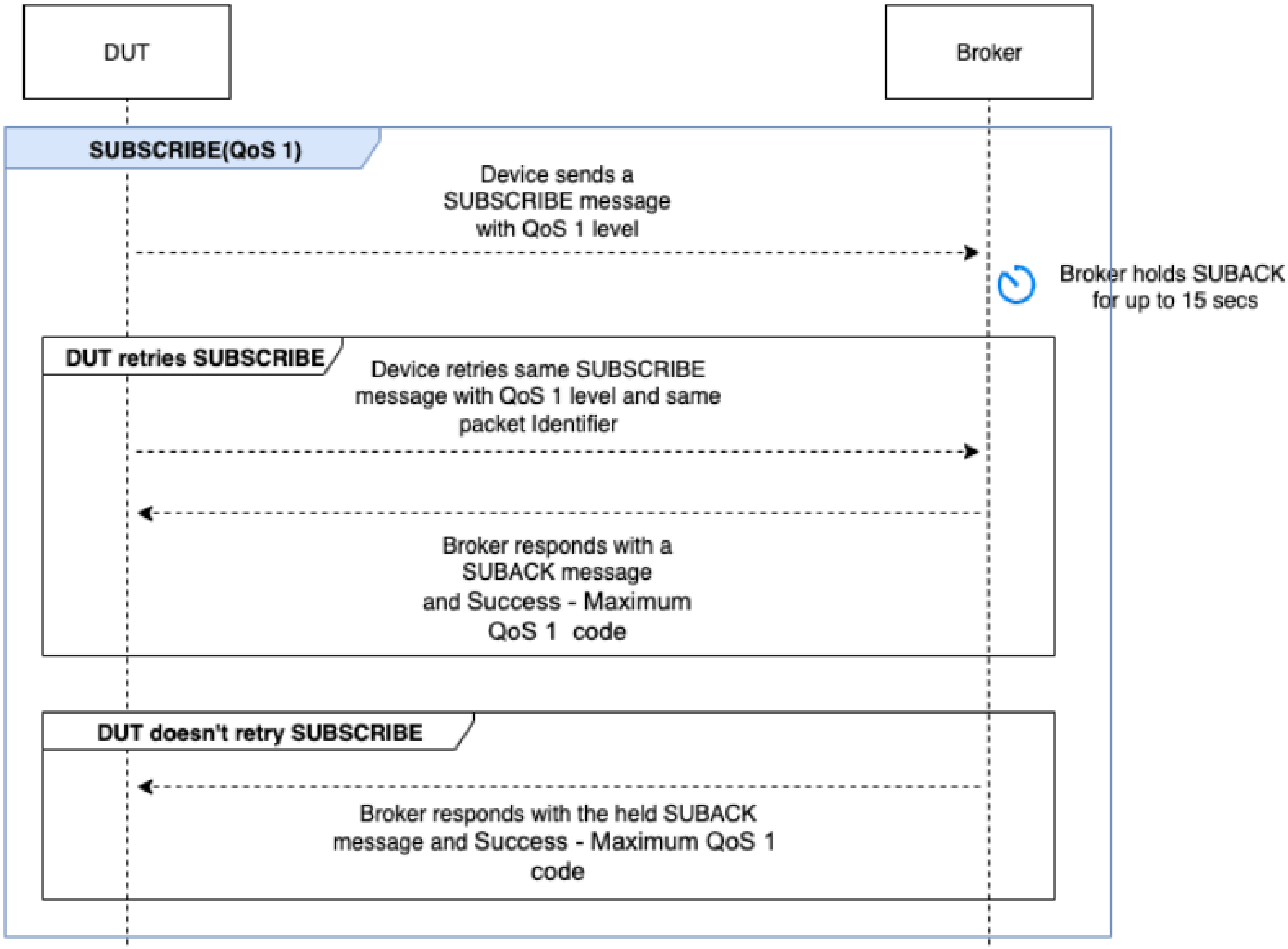
QoS 0

This test case validates if the device successfully sends a `SUBSCRIBE` message to the broker during a subscribe with QoS 0. The test doesn't wait for the device to receive a SUBACK message.



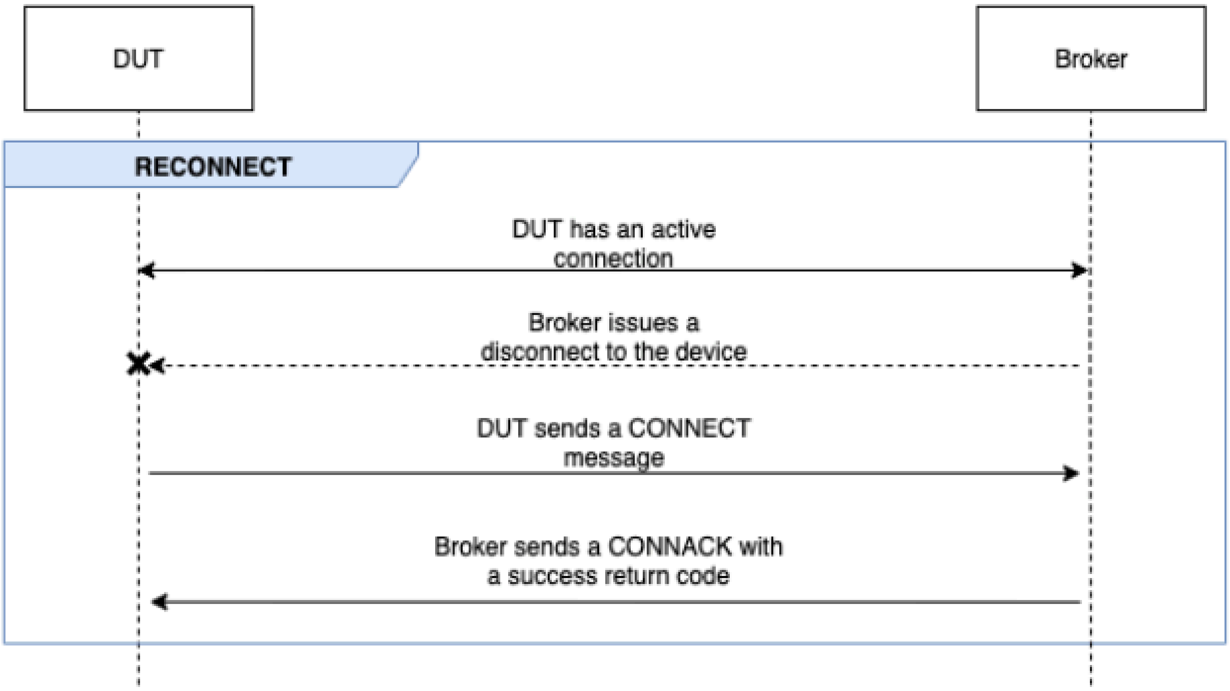
QoS 1

In this test case, the device is expected to send two `SUBSCRIBE` messages to the broker with QoS 1. After the first `SUBSCRIBE` message, the broker waits for up to 15 seconds before it responds. The device must retry the original `SUBSCRIBE` message with the same packet identifier within the 15 second window. If it does, the broker responds with a `SUBACK` message and the test validates. If the device doesn't retry the `SUBSCRIBE`, the original `SUBACK` is sent to the device and the test is marked as **Pass with warnings**, along with a system message. During the test execution, if the device loses connection and reconnects, the test scenario will reset without failing and the device has to perform the test scenario steps again.



RECONNECT

This scenario validates if the device successfully reconnects with the broker after the device is disconnected from a successful connection. Device Advisor won't disconnect the device if it connected more than once previously during the test suite. Instead, it will mark the test as **Pass**.



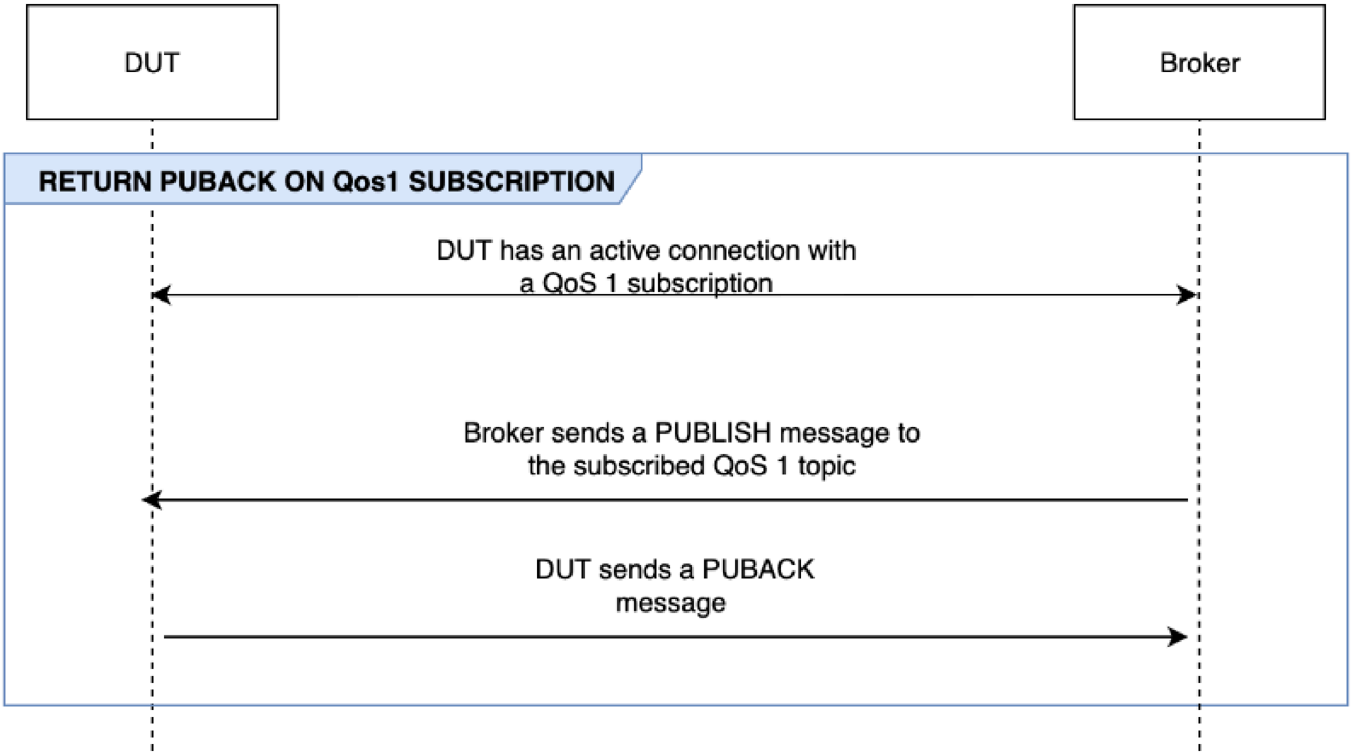
Advanced tests execution

In this phase, the test case runs more complex tests in serial to validate if the device follows best practices. These advanced tests are available for selection and can be opted out if not required. Each advanced test has its own timeout value based on what the scenario demands.

RETURN PUBACK ON QoS 1 SUBSCRIPTION

> **Note**
>
> Only select this scenario if your device is capable of performing QoS 1 subscriptions.

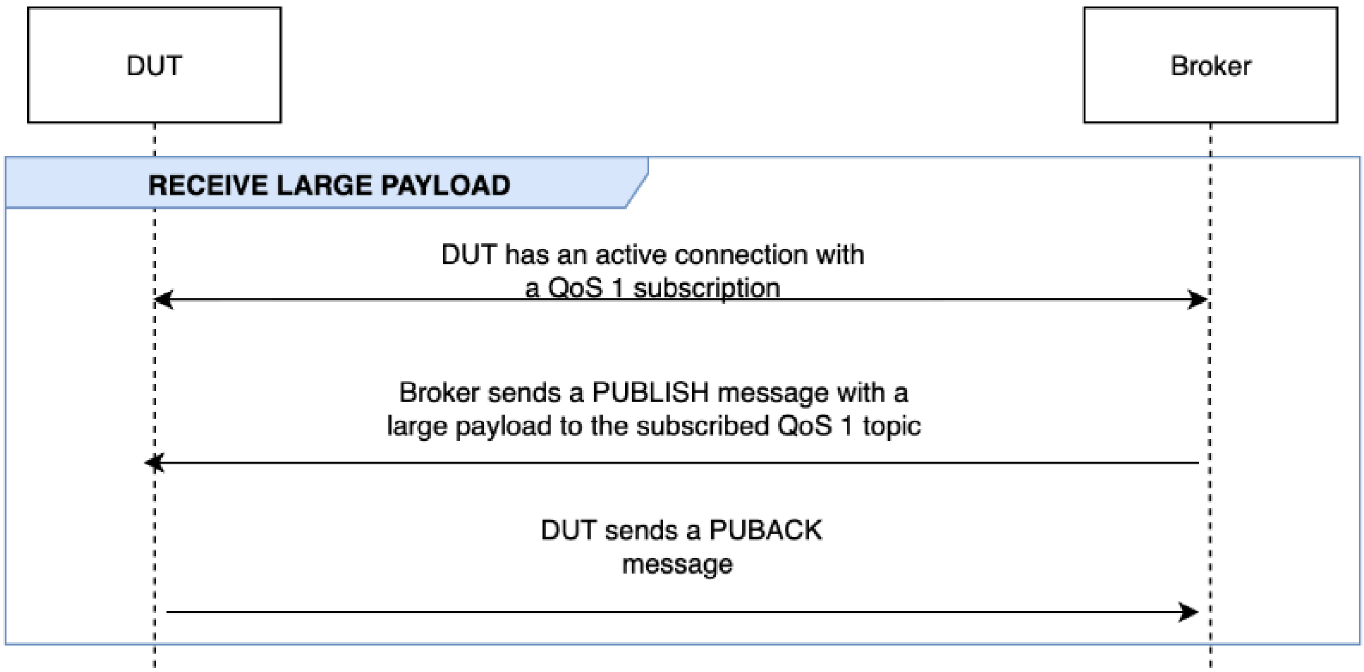
This scenario validates if, after the device subscribes to a topic and receives a `PUBLISH` message from the broker, it returns a `PUBACK` message.



RECEIVE LARGE PAYLOAD

> **Note**
>
> Select this scenario only if your device is capable of performing QoS 1 subscriptions.

This scenario validates if the device responds with a `PUBACK` message after receiving a `PUBLISH` message from the broker for a QoS 1 topic with a large payload. The format of the expected payload can be configured using the `LONG_PAYLOAD_FORMAT` option.



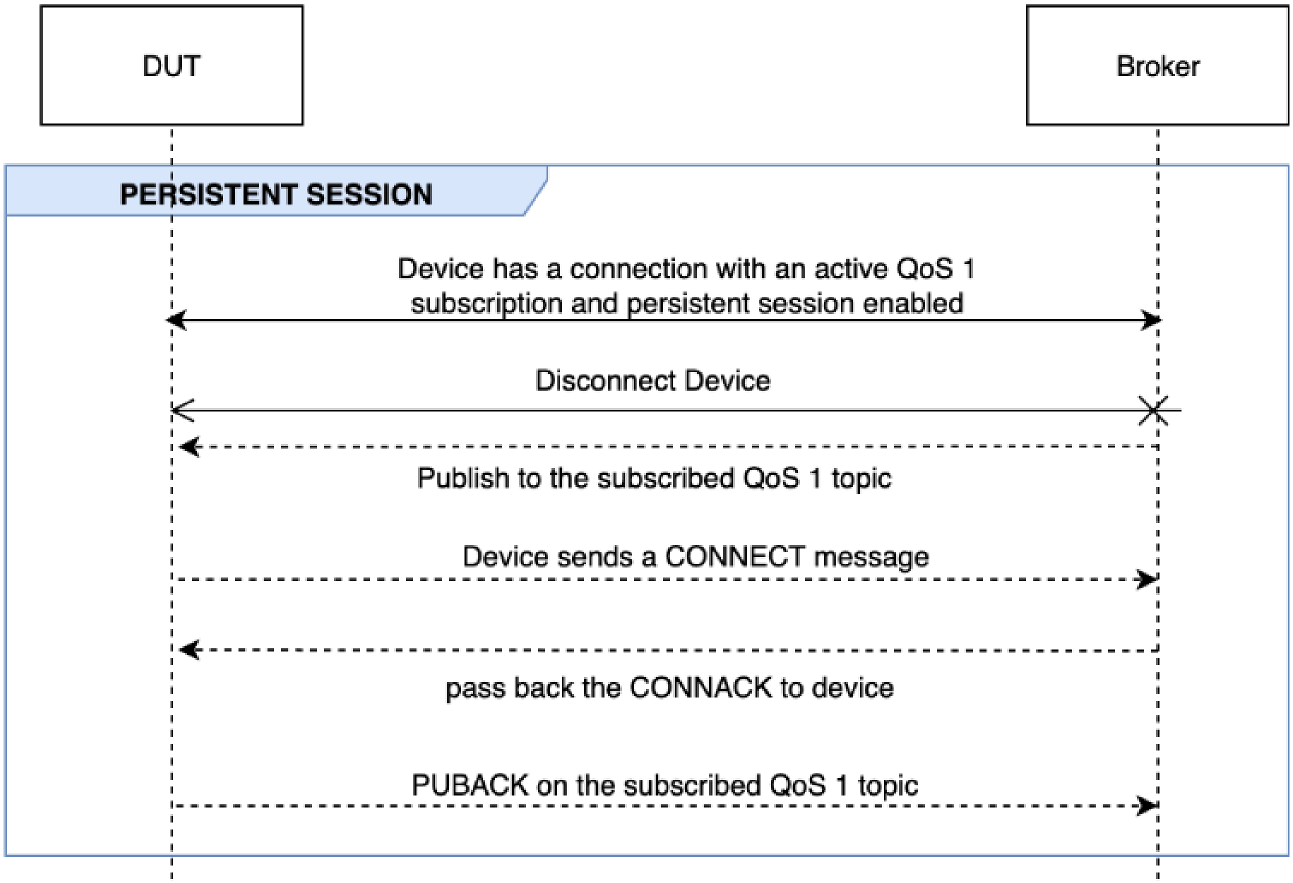
PERSISTENT SESSION

> **Note**
>
> Select this scenario only if your device is capable of performing QoS 1 subscriptions and can maintain a persistent session.

This scenario validates the device behavior in maintaining persistent sessions. The test validates when the following conditions are met:

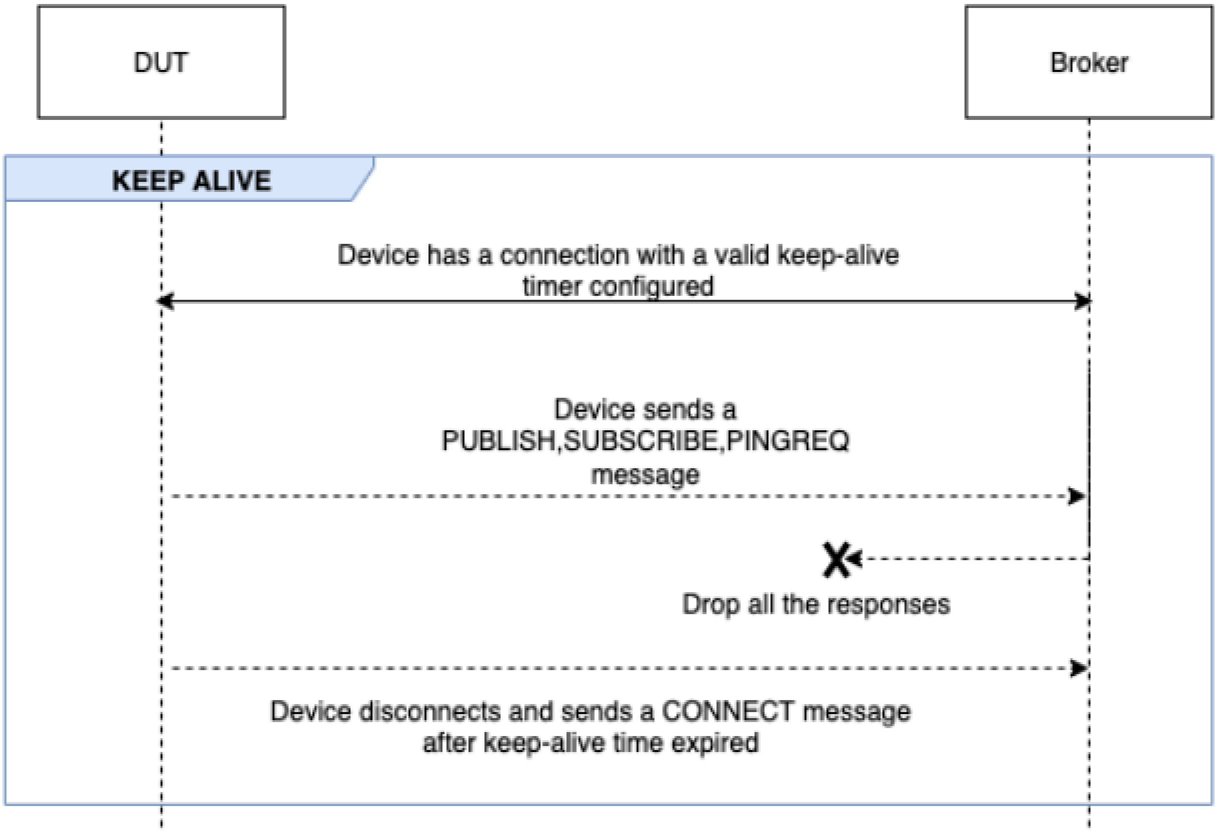
- The device connects to the broker with an active QoS 1 subscription and persistent sessions enabled.
- The device successfully disconnects from the broker during the session.
- The device reconnects to the broker and resumes subscriptions to its trigger topics without explicitly resubscribing to those topics.
- The device successfully receives messages stored by the broker for its subscribed topics and runs as expected.

For more information on AWS IoT Persistent Sessions, see [Using MQTT persistent sessions](#).



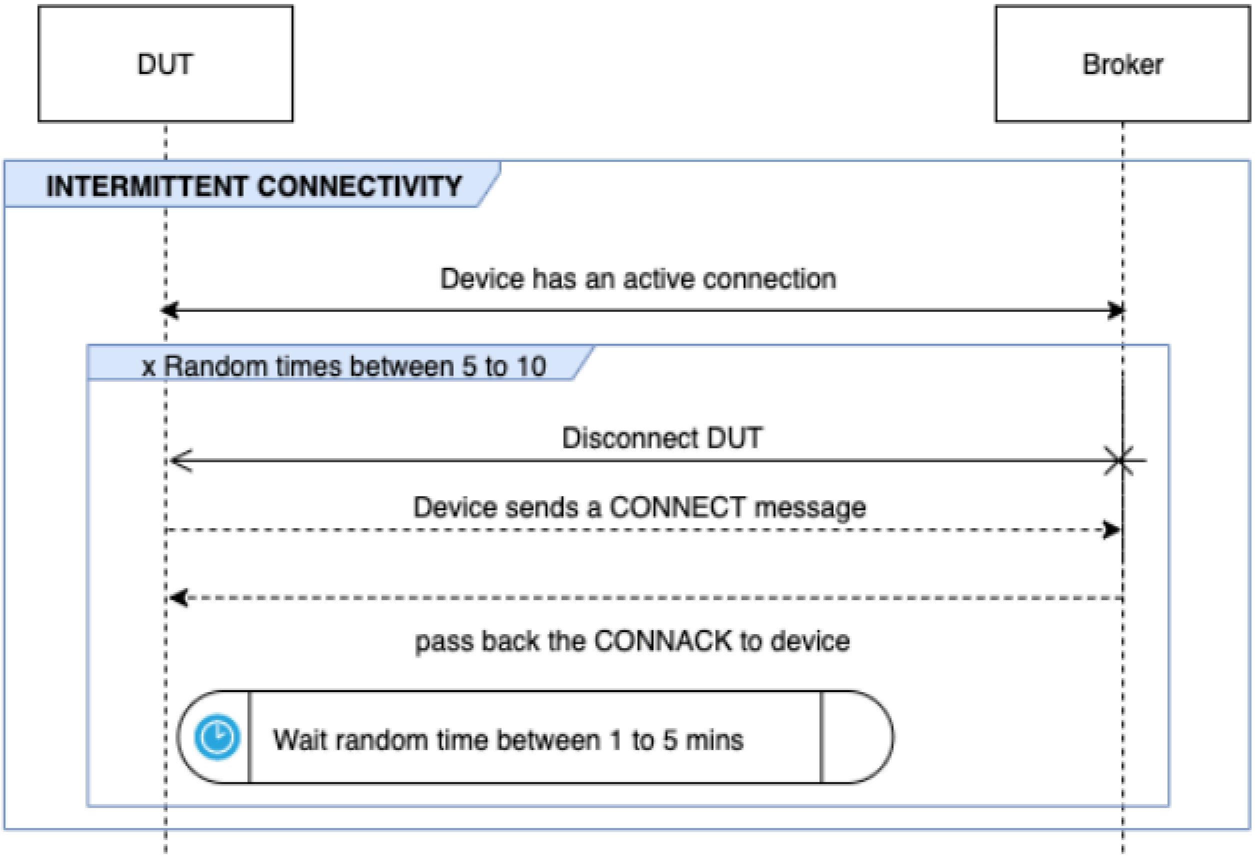
KEEP ALIVE

This scenario validates if the device successfully disconnects after it doesn't receive a ping response from the broker. The connection must have a valid keep-alive timer configured. As part of this test, the broker blocks all responses sent for `PUBLISH`, `SUBSCRIBE`, and `PINGREQ` messages. It also validates if the device under test disconnects the MQTT connection.



INTERMITTENT CONNECTIVITY

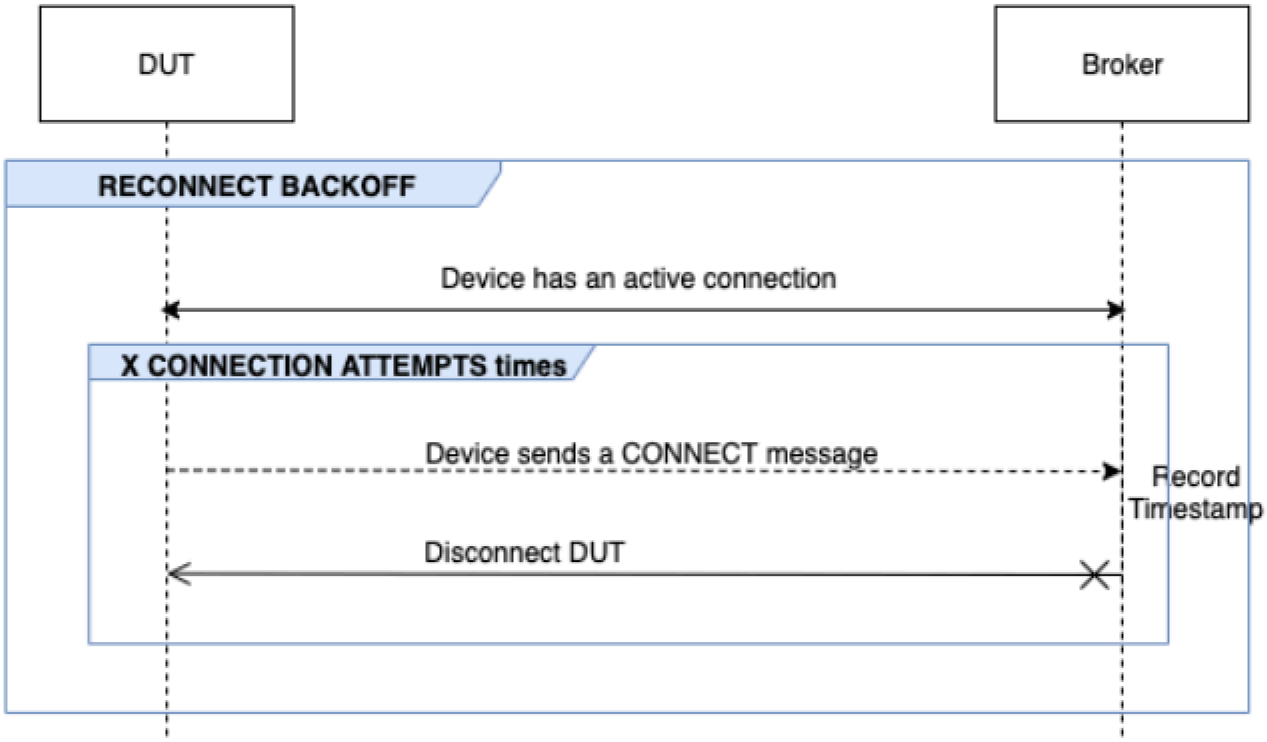
This scenario validates if the device can connect back to the broker after the broker disconnects the device at random intervals for a random period of time.



RECONNECT BACKOFF

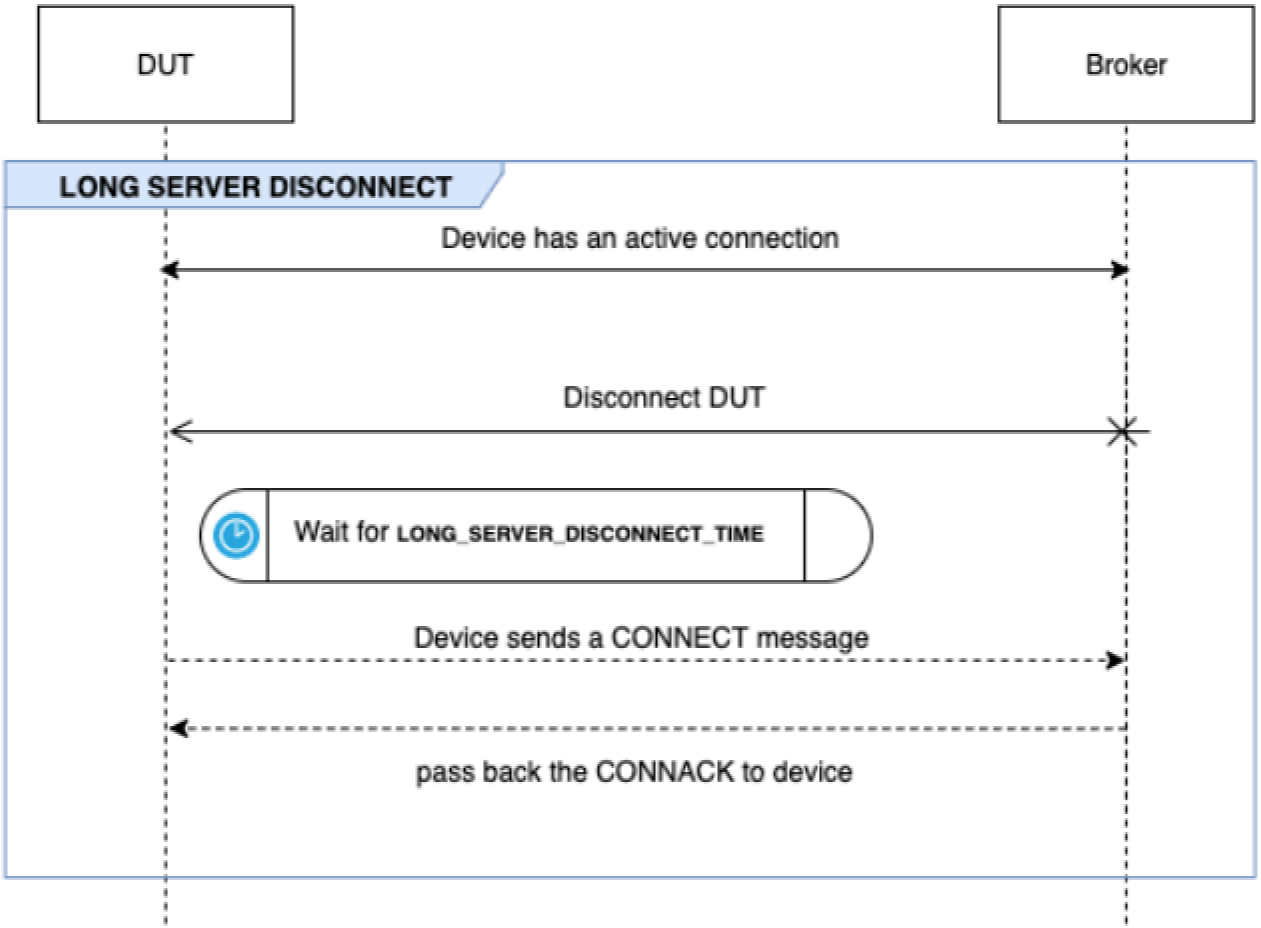
This scenario validates if the device has a backoff mechanism implemented when the broker disconnects from it multiple times. Device Advisor reports the backoff type as exponential, jitter, linear or constant. The number of backoff attempts is configurable using the `BACKOFF_CONNECTION_ATTEMPTS` option. The default value is 5. The value is configurable between 5 and 10.

To pass this test, we recommend implementing the [Exponential Backoff And Jitter] mechanism on the device under test.



LONG SERVER DISCONNECT

This scenario validates if the device can successfully reconnect after the broker disconnects the device for a long period of time (up to 120 minutes). The time for server disconnection can be configured using the `LONG_SERVER_DISCONNECT_TIME` option. The default value is 120 minutes. This value is configurable from 30 to 120 minutes.



Additional execution time

The additional execution time is the time the test waits after completing all the above tests and before ending the test case. Customers use this additional time period to monitor and log all communications between the device and the broker. The additional execution time can be configured using the `ADDITIONAL_EXECUTION_TIME` option. By default, this option is set to 0 minutes and can be 0 to 120 minutes.

MQTT long duration test configuration options

All configuration options provided for the MQTT long duration test are optional. The following options are available:

OPERATIONS

The list of operations that the device performs, such as `CONNECT`, `PUBLISH` and `SUBSCRIBE`. The test case runs scenarios based on the specified operations. Operations that aren't specified are assumed valid.

```
{
"OPERATIONS": ["PUBLISH", "SUBSCRIBE"]
//by default the test assumes device can CONNECT
}
```

SCENARIOS

Based on the operations selected, the test case runs scenarios to validate the device's behavior. There are two types of scenarios:

- **Basic Scenarios** are simple tests that validate if the device can perform the operations selected above as part of the configuration. These are pre-selected based on the operations specified in the configuration. No more input is required in the configuration.
- **Advanced Scenarios** are more complex scenarios that are performed against the device to validate if the device follows best practices when met with real world conditions. These are optional and can be passed as an array of scenarios to the configuration input of the test suite.

```

{
    "SCENARIOS": [      // list of advanced scenarios
                "PUBACK_QOS_1",
                "RECEIVE_LARGE_PAYLOAD",
                "PERSISTENT_SESSION",
                "KEEP_ALIVE",
                "INTERMITTENT_CONNECTIVITY",
                "RECONNECT_BACK_OFF",
                "LONG_SERVER_DISCONNECT"
    ]
}
```

BASIC_TESTS_EXECUTION_TIME_OUT:

The maximum time the test case will wait for all the basic tests to complete. The default value is 60 minutes. This value is configurable from 30 to 120 minutes.

LONG_SERVER_DISCONNECT_TIME:

The time taken for the test case to disconnect and reconnect the device during the Long Server Disconnect test. The default value is 60 minutes. This value is configurable from 30 to 120 minutes.

ADDITIONAL_EXECUTION_TIME:

Configuring this option provides a time window after all the tests are completed, to monitor events between the device and broker. The default value is 0 minutes. This value is configurable from 0 to 120 minutes.

BACKOFF_CONNECTION_ATTEMPTS:

This option configures the number of times the device is disconnected by the test case. This is used by the Reconnect Backoff test. The default value is 5 attempts. This value is configurable from 5 to 10.

LONG_PAYLOAD_FORMAT:

The format of the message payload that the device expects when the test case publishes to a QoS 1 topic subscribed by the device.

API test case definition:

```
{
"tests":[
   {
      "name":"my_mqtt_long_duration_test",
      "configuration": {
         // optional
         "OPERATIONS": ["PUBLISH", "SUBSCRIBE"],
         "SCENARIOS": [
            "LONG_SERVER_DISCONNECT",
            "RECONNECT_BACK_OFF",
            "KEEP_ALIVE",
            "RECEIVE_LARGE_PAYLOAD",
            "INTERMITTENT_CONNECTIVITY",
            "PERSISTENT_SESSION",
         ],
         "BASIC_TESTS_EXECUTION_TIMEOUT": 60, // in minutes (60 minutes by default)
         "LONG_SERVER_DISCONNECT_TIME": 60,   // in minutes (120 minutes by default)
         "ADDITIONAL_EXECUTION_TIME": 60,     // in minutes (0 minutes by default)
```

```
          "BACKOFF_CONNECTION_ATTEMPTS": "5",
          "LONG_PAYLOAD_FORMAT":"{"message":"${payload}"}"
       },
       "test":{
          "id":"MQTT_Long_Duration",
          "version":"0.0.0"
       }
    }
 ]
}
```

MQTT long duration test case summary log

The MQTT long duration test case runs for longer duration than regular test cases. A separate summary log is provided, which lists important events such as device connections, publish, and subscribe during the run. Details include what was tested, what was not tested and what failed. At the end of the log, the test includes a summary of all the events that happened during the test case run. This includes:

- *Keep Alive timer configured on the device.*
- *Persistent session flag configured on the device.*
- *The number of device connections during the test run.*
- *The device reconnection backoff type, if validated for the reconnect backoff test.*
- *The topics the device published to, during the test case run.*
- *The topics the device subscribed to, during the test case run.*

AWS IoT Core Device Location

Before using the AWS IoT Core Device Location feature, review the Terms and Conditions for this feature. Note that AWS may transmit your geolocation search request parameters, such as the location data used to run searches, and other information to your chosen third party data provider, which may be outside of the AWS Region that you are currently using. The third party provider and the solver to be used is chosen based on the input payload received. For more information, see AWS Service Terms.

Use AWS IoT Core Device Location to test the location of your IoT devices using third-party solvers. *Solvers* are algorithms provided by third-party vendors that resolve measurement data and estimate the location of your device. By identifying the location of your devices, you can track and debug them in the field to troubleshoot any issues.

The measurement data collected from various sources is resolved, and the geolocation information is reported as a GeoJSON payload. The GeoJSON format is a format that's used to encode geographic data structures. The payload contains the latitude and longitude coordinates of your device location, which are based on the World Geodetic System coordinate system (WGS84).

Topics

Measurement types and solvers

AWS IoT Core Device Location partners with third-party vendors to resolve the measurement data and to provide an estimated device location. The following table shows the measurement types and the third-party location solvers, and information about supported devices. For information about LoRaWAN devices and configuring device location for them, see Configuring position of LoRaWAN resources.

> **Note**
>
> General IoT devices and Sidewalk devices can use the device location MQTT topics to obtain the location information. For Wi-Fi, Cellular, and IP address measurement types, if the devices publish the measurement data to the reserved topics in the defined GeoJSON format, AWS IoT Core Device Location can resolve the location of the device. For GNSS measurement type, the device must have the LR11xx chip to scan the measurement data for obtaining the resolved location information using the GNSS solver. For information about obtaining location information for LoRaWAN devices, see Configuring position for LoRaWAN resources in the *AWS IoT Wireless documentation*.

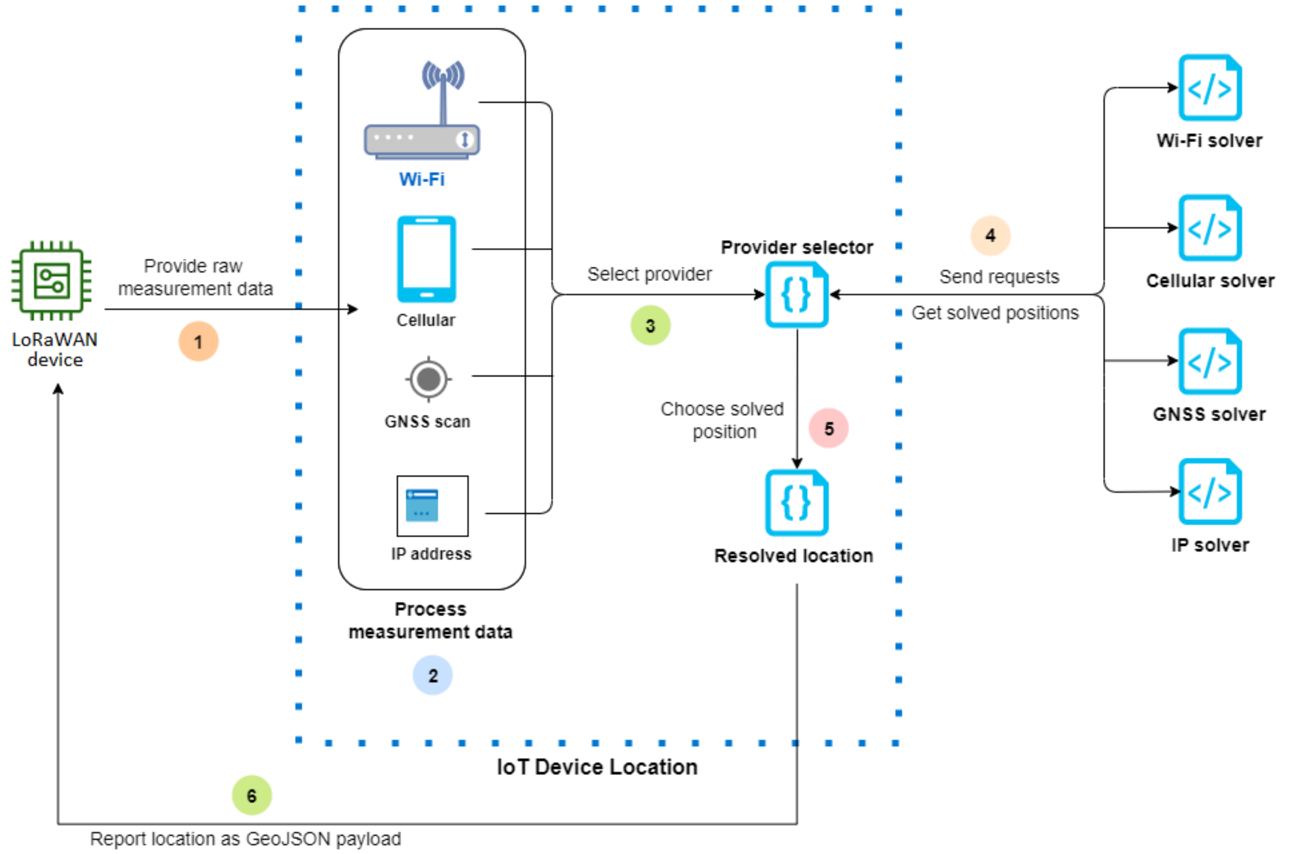
Measurement types and solvers

Measurement type	Third-party solvers	Supported devices
Wi-Fi access points	Wi-Fi based solver	General IoT devices, LoRaWAN, and Amazon Sidewalk devices
Cellular radio towers: GSM, LTE, CDMA, SCDMA, WCMDA, and TD-SCDMA data	Cellular based solver	General IoT devices, LoRaWAN, and Amazon Sidewalk devices
IP address	IP reverse lookup solver	Any IoT device that connects over TCP/IP
GNSS scan data (NAV messages)	GNSS solver	General IoT devices, LoRaWAN, and Amazon Sidewalk devices
Bluetooth Low Energy (BLE)	BLE based solver	Amazon Sidewalk devices

For more information about the location solvers and examples that show the device payload for the various measurement types, see Location solvers and device payload.

How AWS IoT Core Device Location works

The following diagram shows how AWS IoT Core Device Location collects measurement data and resolves the location information of your devices.



The following steps show how AWS IoT Core Device Location works.

1. **Receive measurement data**

   The raw measurement data related to your device location is first sent from the device. The measurement data is specified as a JSON payload.

2. **Process measurement data**

   The measurement data is processed, and AWS IoT Core Device Location chooses the measurement data to be used, which can be Wi-Fi, cellular, GNSS scan, or IP address information.

3. **Choose solver**

   The third-party solver is chosen based on the measurement data. For example, if the measurement data contains Wi-Fi and IP address information, it chooses the Wi-Fi solver and the IP reverse lookup solver.

4. **Obtain resolved location**

   An API request is sent to the solver providers requesting to resolve the location. AWS IoT Core Device Location then gets the estimated geolocation information from the solvers.

5. **Choose resolved location**

   The resolved location information and its accuracy is compared, and AWS IoT Core Device Location chooses the geolocation results with the highest accuracy.

6. **Output location information**

   The geolocation information is sent to you as a GeoJSON payload. The payload contains the WGS84 geo coordinates, the accuracy information, confidence levels, and the timestamp at which the resolved location was obtained.

How to use AWS IoT Core Device Location

The following steps show how to use AWS IoT Core Device Location.

1. **Provide measurement data**

   Specify the raw measurement data related to the location of your device as a JSON payload. To retrieve the payload measurement data, go to your device logs, or use CloudWatch Logs, and copy the payload data information. The JSON payload must contain one or more types of data measurement. For examples that show the payload format for various solvers, see Location solvers and device payload.

2. **Resolve location information**

   Using the Device Location page in the AWS IoT console or the GetPositionEstimate API operation, pass the payload measurement data and resolve the device location. AWS IoT Core Device Location then chooses the solver with the highest accuracy and reports the device location. For more information, see Resolving location of IoT devices.

3. **Copy location information**

   Verify the geolocation information that was resolved by AWS IoT Core Device Location and reported as a GeoJSON payload. You can copy the payload for use with your applications and other AWS services. For example, you can send your geographical location data to Amazon Location Service using the Location AWS IoT rule action.

The following topics show how to use AWS IoT Core Device Location and examples of device location payload.

- Resolving location of IoT devices
- Location solvers and device payload

Resolving location of IoT devices

Use AWS IoT Core Device Location to decode the measurement data from your devices, and resolve the device location using third-party solvers. The resolved location is generated as a GeoJSON payload with the geo coordinates and accuracy information. You can resolve the location of your device from the AWS IoT console, the AWS IoT Wireless API, or AWS CLI.

Topics

- Resolving device location (console)
- Resolving device location (API)
- Troubleshooting errors when resolving the location

Resolving device location (console)

To resolve the device location (console)

1. Go to the Device Location page in the AWS IoT console.
2. Obtain the payload measurement data from your device logs or from CloudWatch Logs, and enter it in the **Resolve position via payload** section.

   The following code shows a sample JSON payload. The payload contains cellular and Wi-Fi measurement data. If your payload contains additional types of measurement data, the solver with the best accuracy will be used. For more information and payload examples, see the section called “Location solvers and device payload”.

   > **Note**
   >
   > The JSON payload must contain at least one type of measurement data.

```
{
    "Timestamp": 1664313161,
    "Ip":{
        "IpAddress": "54.240.198.35"
    },
    "WiFiAccessPoints": [{
        "MacAddress": "A0:EC:F9:1E:32:C1",
        "Rss": -77
    }],
    "CellTowers": {
        "Gsm": [{
            "Mcc": 262,
            "Mnc": 1,
            "Lac": 5126,
            "GeranCid": 16504,
            "GsmLocalId": {
                "Bsic": 6,
                "Bcch": 82
            },
            "GsmTimingAdvance": 1,
            "RxLevel": -110,
            "GsmNmr": [{
                "Bsic": 7,
                "Bcch": 85,
                "RxLevel": -100,
                "GlobalIdentity": {
                    "Lac": 1,
                    "GeranCid": 1
                }
            }]
        }],
        "Wcdma": [{
            "Mcc": 262,
            "Mnc": 7,
            "Lac": 65535,
            "UtranCid": 14674663,
            "WcdmaNmr": [{
                    "Uarfcndl": 10786,
                    "UtranCid": 14674663,
                    "Psc": 149
                },
                {
```

```
                    "Uarfcndl": 10762,
                    "UtranCid": 14674663,
                    "Psc": 211
                }
            ]
        }],
        "Lte": [{
            "Mcc": 262,
            "Mnc": 2,
            "EutranCid": 2898945,
            "Rsrp": -50,
            "Rsrq": -5,
            "LteNmr": [{
                    "Earfcn": 6300,
                    "Pci": 237,
                    "Rsrp": -60,
                    "Rsrq": -6,
                    "EutranCid": 2898945
                },
                {
                    "Earfcn": 6300,
                    "Pci": 442,
                    "Rsrp": -70,
                    "Rsrq": -7,
                    "EutranCid": 2898945
                }
            ]
        }]
    }
}
```


3. To resolve the location information, choose **Resolve**.

   The location information is of type blob and returned as a payload that uses the GeoJSON format, which is a format used for encoding geographical data structures. The payload contains:

   - The WGS84 geo coordinates, which include the latitude and longitude information. It might also include an altitude information.
   - The type of location information reported, such as **Point**. A point location type represents the location as a WGS84 latitude and longitude, encoded as a GeoJSON point.

- The horizontal and vertical accuracy information, which indicates the difference, in meters, between the location information estimated by the solvers and the actual device location.
- The confidence level, which indicates the uncertainty in the location estimate response. The default value is 0.68, which indicates a 68% probability that the actual device location is within the uncertainty radius of the estimated location.
- The city, state, country, and postal code where the device is located. This information will be reported only when the IP reverse lookup solver is used.
- The timestamp information, which corresponds to the date and time at which the location was resolved. It uses the Unix timestamp format.

The following code shows a sample GeoJSON payload returned by resolving the location.

> **Note**
>
> If AWS IoT Core Device Location reports errors when attempting to resolve the location, you can troubleshoot the errors and resolve the location. For more information, see Troubleshooting errors when resolving the location.

```
{
    "coordinates": [
        13.376076698303223,
        52.51823043823242
    ],
    "type": "Point",
    "properties": {
        "verticalAccuracy": 45,
        "verticalConfidenceLevel": 0.68,
        "horizontalAccuracy": 303,
        "horizontalConfidenceLevel": 0.68,
        "country": "USA",
        "state": "CA",
        "city": "Sunnyvalue",
        "postalCode": "91234",
        "timestamp": "2022-11-18T12:23:58.189Z"
    }
}
```

4. Go to the **Resource location** section and verify the geolocation information reported by AWS IoT Core Device Location . You can copy the payload for use with other applications and AWS services. For example, you can use the Location to send your geographical location data to Amazon Location Service.

Resolving device location (API)

To resolve the device location using the AWS IoT Wireless API, use the GetPositionEstimate API operation or the get-position-estimate CLI command. Specify the payload measurement data as input, and run the API operation to resolve the device location.

> **Note**
>
> The `GetPositionEstimate` API operation doesn't store any device or state information and can't be used retrieve historical location data. It performs a one-time operation that resolves the measurement data and produces the estimated location. To retrieve the location information, you must specify the payload information every time you perform this API operation.

The following command shows an example of how to resolve the location using this API operation.

> **Note**
>
> When running the `get-position-estimate` CLI command, you must specify the output JSON file as the first input. This JSON file will store the estimated location information obtained as response from the CLI in GeoJSON format. For example, the following command stores the location information in the *`locationout.json`* file.

```
aws iotwireless get-position-estimate locationout.json \
    --ip IpAddress=""54.240.198.35"" \
    --wi-fi-access-points \
        MacAddress="A0:EC:F9:1E:32:C1",Rss=-75 \
        MacAddress="A0:EC:F9:15:72:5E",Rss=-67
```

This example includes both Wi-Fi access points and IP address as the measurement types. AWS IoT Core Device Location chooses between the Wi-Fi solver and the IP reverse lookup solver, and it selects the solver with the higher accuracy.

The resolved location is returned as a payload that uses the GeoJSON format, which is a format used for encoding geographical data structures. It is then stored in the *locationout.json* file. The payload contains the WGS84 latitude and longitude coordinates, accuracy and confidence level information, the location data type, and the timestamp at which the location was resolved.


```
{
    "coordinates": [
        13.37704086303711,
        52.51865005493164
    ],
    "type": "Point",
    "properties": {
        "verticalAccuracy": 707,
        "verticalConfidenceLevel": 0.68,
        "horizontalAccuracy": 389,
        "horizontalConfidenceLevel": 0.68,
        "country": "USA",
        "state": "CA",
        "city": "Sunnyvalue",
        "postalCode": "91234",
        "timestamp": "2022-11-18T14:03:57.391Z"
    }
}
```


Troubleshooting errors when resolving the location

When you attempt to resolve the location, you might see any of the following error codes. AWS IoT Core Device Location might generate an error when using the `GetPositionEstimate` API operation, or else refer to the line number corresponding to the error in the AWS IoT console.

- **400 error**

  This error indicates that the format of the device payload JSON can't be validated by AWS IoT Core Device Location. The error might occur because:

  - The JSON measurement data is formatted incorrectly.
  - The payload contains only the timestamp information.

  - The measurement data parameters, such as the IP address, are not valid.

  To resolve this error, check whether your JSON is formatted correctly and contains data from one or more measurement types as input. If the IP address is invalid, for information about how you can provide a valid IP address to resolve the error, see IP reverse lookup solver.

- **403 error**

  This error indicates that you don't have the permissions to perform the API operation or to use the AWS IoT console to retrieve the device location. To resolve this error, verify that you have the required permissions to perform this action. This error might occur if your AWS Management Console session or your AWS CLI session token have expired. To resolve this error, refresh the session token to use the AWS CLI, or log out of the AWS Management Console and then log in using your credentials.

- **404 error**

  This error indicates that no location information was found or solved by AWS IoT Core Device Location. The error might occur due to cases such as insufficient data in the measurement data input. For example:

  - The MAC address or cellular tower information is not sufficient.
  - The IP address is not available to look up and retrieve the location.
  - The GNSS payload is not sufficient.

  To resolve the error in such cases, check whether your measurement data contains sufficient information required to resolve the device location.

- **500 error**

  This error indicates that an internal server exception occurred when AWS IoT Core Device Location attempted to resolve the location. To attempt to fix this error, refresh the session and retry sending the measurement data to be resolved.

Resolving device location using AWS IoT Core Device Location MQTT topics

You can use reserved MQTT topics to get the latest location information for your devices with the AWS IoT Core Device Location feature.

Format of device location MQTT topics

Reserved topics for AWS IoT Core Device Location use the following prefix:

`$aws/device_location/{customer_device_id}/`

To create a complete topic, first replace *customer_device_id* with your unique ID that you use for identifying your device. We recommend that you specify the `WirelessDeviceId`, such as for LoRaWAN and Sidewalk devices, and *thingName*, if your device is registered as an AWS IoT thing. You then append the topic with the topic stub, such as `get_position_estimate` or `get_position_estimate/accepted` as shown in the following section.

> **Note**
>
> The *{customer_device_id}* can only contain letters, numbers, and dashes. When subscribing to device location topics, you can only use the plus sign (+) as a wildcard character. For example, you can use the + wildcard for the *{customer_device_id}* to obtain the location information for your devices. When you subscribe to the topic `$aws/device_location/+/get_position_estimate/accepted`, a message will be published with the location information for devices that match any device ID if it was successfully resolved.

The following are the reserved topics used to interact with AWS IoT Core Device Location.

Device location MQTT topics

Topic	Allowed operations	Description
$aws/device_location/*customer_device_id* /get_position_estimate	Publish	A device publishes to this topic to get the scanned raw measurement data to be resolved by AWS IoT Core Device Location.
$aws/device_location/*customer_device_id* /get_position_estimate/accepted	Subscribe	AWS IoT Core Device Location publishes the location information to this topic when it successfully resolves the device location.

Topic	Allowed operations	Description
$aws/device_location/*customer_device_id*/get_position_estimate/rejected	Subscribe	AWS IoT Core Device Location publishes the error information to this topic when it fails to resolve the device location.

Policy for device location MQTT topics

To receive messages from device location topics, your device must use a policy that allows it to connect to the AWS IoT device gateway and subscribe to the MQTT topics.

The following is an example of the policy required for receiving messages for the various topics.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/
device-123/get_position_estimate"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/
device-123/get_position_estimate/accepted",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/
device-123/get_position_estimate/rejected"
            ]
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
device_location/device-123/get_position_estimate/accepted",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
device_location/device-123/get_position_estimate/rejected"
            ]
        }
    ]
}
```

Device location topics and payload

The following shows the AWS IoT Core Device Location topics, the format of their message payload, and an example policy for each topic.

Topics

- /get_position_estimate
- /get_position_estimate/accepted
- /get_position_estimate/rejected

/get_position_estimate

Publish a message to this topic to get the raw measurement data from the device to be resolved by AWS IoT Core Device Location.

```
$aws/device_location/customer_device_id/get_position_estimate
```

AWS IoT Core Device Location responds by publishing to either /get_position_estimate/accepted or /get_position_estimate/rejected.

Note

The message published to this topic must be a valid JSON payload. If the input message is not in valid JSON format, you won't get any response. For more information, see Message payload.

Message payload

The message payload format follows a similar structure as the AWS IoT Wireless API operation request body, `GetPositionEstimate`. It contains:

- An optional `Timestamp` string, which corresponds to the date and time the location was resolved. The `Timestamp` string can have a minimum length of 1 and maximum length of 10.
- An optional `MessageId` string, which can be used to map the request to the response. If you specify this string, the message published to the `get_position_estimate/accepted` or `get_position_estimate/rejected` topics will contain this `MessageId`. The `MessageID` string can have a minimum length of 1 and maximum length of 256.
- The measurement data from the device that contains one or more of the following measurement types:
  - `WiFiAccessPoint`
  - `CellTowers`
  - `IpAddress`
  - `Gnss`

The following shows a sample message payload.

```
{
    "Timestamp": "1664313161",
    "MessageId": "ABCD1",
    "WiFiAccessPoints":  [
        {
            "MacAddress": "A0:EC:F9:1E:32:C1",
            "Rss": -66
        }
    ],
    "Ip":{
        "IpAddress": "54.192.168.0"
```

```
    },
    "Gnss":{
      "Payload":"8295A614A2029517F4F77C0A7823B161A6FC57E25183D96535E3689783F6CA48",
      "CaptureTime":1354393948
    }
}
```

Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/
device-123/get_position_estimate"
            ]
        }
    ]
}
```

/get_position_estimate/accepted

AWS IoT Core Device Location publishes a response to this topic when returning the resolved location information for your device. The location information is returned in GeoJSON format.

```
$aws/device_location/customer_device_id/get_position_estimate/accepted
```

The following shows the message payload and an example policy.

Message payload

The following is an example of the message payload in GeoJSON format. If you specified a `MessageId` in your raw measurement data and AWS IoT Core Device Location resolved the location information successfully, then the message payload returns the same `MessageId` information.

```
{
    "coordinates": [
        13.37704086303711,
        52.51865005493164
    ],
    "type": "Point",
    "properties": {
        "verticalAccuracy": 707,
        "verticalConfidenceLevel": 0.68,
        "horizontalAccuracy": 389,
        "horizontalConfidenceLevel": 0.68,
        "country": "USA",
        "state": "CA",
        "city": "Sunnyvalue",
        "postalCode": "91234",
        "timestamp": "2022-11-18T14:03:57.391Z",
        "messageId": "ABCD1"
    }
}
```


Example policy

The following is an example of the required policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
device_location/device-123/get_position_estimate/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
```

```
                "Resource": [
                    "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/
device-123/get_position_estimate/accepted"
                ]
            }
        ]
}
```

/get_position_estimate/rejected

AWS IoT Core Device Location publishes an error response to this topic when it fails to resolve the device location.

```
$aws/device_location/customer_device_id/get_position_estimate/rejected
```

The following shows the message payload and example policy. For information about the errors, see Troubleshooting errors when resolving the location.

Message payload

The following is an example of the message payload that provides the error code and message, which indicates why AWS IoT Core Device Location failed to resolve the location information. If you specified a `MessageId` when providing your raw measurement data and AWS IoT Core Device Location failed to resolve the location information, then the same `MessageId` information will be returned in the message payload.

```
{
    "errorCode": 500,
    "errorMessage":"Internal server error",
    "messageId": "ABCD1"
}
```

Example policy

The following is an example of the required policy:

```
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/device_location/
device-123/get_position_estimate/rejected"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topic/$aws/device_location/device-123/
get_position_estimate/rejected"
      ]
    }
  ]
}
```

Location solvers and device payload

Location solvers are algorithms that can be used to resolve the location of your IoT devices. AWS IoT Core Device Location supports the following location solvers. You'll see examples of the JSON payload format for these measurement types, the devices supported by the solver, and how the location is resolved.

To resolve the device location, specify one or more of these measurement data types. A single, resolved location will be returned for all measurement data combined.

Topics

Wi-Fi based solver

Use the Wi-Fi based solver to resolve the location using the scan information from Wi-Fi access points. The solver supports the WLAN technology, and it can be used to compute the device location for general IoT devices connecting over TCP/IP, LoRaWAN wireless devices, and Amazon Sidewalk enabled devices.

The LoRaWAN devices must have the LoRa Edge chipset, which can decode the incoming Wi-Fi scan information. LoRa Edge is an ultra-low power platform that integrates a long-range LoRa transceiver, multi-constellation GNSS scanner, and passive Wi-Fi MAC scanner targeting geolocation applications. When an uplink message is received from the device, the Wi-Fi scan data is sent to AWS IoT Core Device Location, and the location is estimated based on the Wi-Fi scan results. The decoded information is then passed to the Wi-Fi based solver to retrieve the location information.

To learn more about resolving location data for Amazon Sidewalk enabled devices, visit AWS IoT Core for Amazon Sidewalk developer guide.

Wi-Fi based solver payload example

The following code shows an example of the JSON payload from the device that contains the measurement data. When AWS IoT Core Device Location receives this data as input, it sends an HTTP request to the solver provider to resolve the location information. To retrieve the information, specify values for the MAC Address and RSS (received signal strength). To do this, either provide the JSON payload using this format, or use the WiFiAccessPoints object parameter of the GetPositionEstimate API operation.

```
{
    "Timestamp": 1664313161,    // optional
    "WiFiAccessPoints": [
        {
            "MacAddress": "A0:EC:F9:1E:32:C1",  // required
            "Rss": -75                          // required
        }
    ]
}
```

Cellular based solver

You can use the cellular based solver to resolve the location using measurement data obtained from cellular radio towers. The solver supports the following technologies. A single resolved location information is obtained, even if you include measurement data from any or all of these technologies.

- GSM
- CDMA
- WCDMA
- TD-SCDMA
- LTE

Cellular based solver payload examples

The following code shows examples of the JSON payload from the device that contains cellular measurement data. When AWS IoT Core Device Location receives this data as input, it sends an HTTP request to the solver provider to resolve the location information. To retrieve the information, you either provide the JSON payload using this format in the console, or specify values for the CellTowers parameter of the GetPositionEstimate API operation. You can provide the measurement data by specifying values for parameters using any or all of these cellular technologies.

LTE (Long-term evolution)

When you use this measurement data, you must specify information such as the network and country code of the mobile network, and optional additional parameters including information about the local ID. The following code shows an example of the payload format. For more information about these parameters, see LTE object.

```
{
    "Timestamp": 1664313161,           // optional
    "CellTowers": {
        "Lte": [
        {
          "Mcc": int,                     // required
          "Mnc": int,                     // required
```

```
            "EutranCid": int,             // required. Make sure that you use int for
 EutranCid.
            "Tac": int,                   // optional
            "LteLocalId": {               // optional
                "Pci": int,               // required
                "Earfcn": int,            // required
            },
            "LteTimingAdvance": int,      // optional
            "Rsrp": int,                  // optional
            "Rsrq": float,                // optional
            "NrCapable": boolean,         // optional
            "LteNmr": [                   // optional
                  {
                      "Pci": int,         // required
                      "Earfcn": int,      // required
                      "EutranCid": int,   // required
                      "Rsrp": int,        // optional
                      "Rsrq": float       // optional
                  }
              ]
           }
        ]
    }
}
```

GSM (Global System for Mobile Communications)

When you use this measurement data, you must specify information such as the network and country code of the mobile network, the base station information, and optional additional parameters. The following code shows an example of the payload format. For more information about these parameters, see GSM object.

```
{
    "Timestamp": 1664313161,           // optional
    "CellTowers": {
        "Gsm": [
        {
          "Mcc": int,                   // required
          "Mnc": int,                   // required
          "Lac": int,                   // required
          "GeranCid": int,              // required
          "GsmLocalId": {               // optional
              "Bsic": int,              // required
```

```
                "Bcch": int,              // required
            },
            "GsmTimingAdvance": int,      // optional
            "RxLevel": int,               // optional
            "GsmNmr": [                   // optional
              {
                  "Bsic": int,            // required
                  "Bcch": int,            // required
                  "RxLevel": int,         // optional
                  "GlobalIdentity": {
                      "Lac": int,         // required
                      "GeranCid": int     // required
                  }
               }
            ]
        }
    ]
}
```

CDMA (Code-division multiple access)

When you use this measurement data, you must specify information such as the signal power and identification information, the base station information, and optional additional parameters. The following code shows an example of the payload format. For more information about these parameters, see [CDMA object](#).

```
{
    "Timestamp": 1664313161,                // optional
    "CellTowers": {
        "Cdma": [
        {
            "SystemId": int,                 // required
            "NetworkId": int,                // required
            "BaseStationId": int,            // required
            "RegistrationZone": int,         // optional
            "CdmaLocalId": {                 // optional
              "PnOffset": int,               // required
              "CdmaChannel": int,            // required
            },
            "PilotPower": int,               // optional
            "BaseLat": float,                // optional
            "BaseLng": float,                // optional
            "CdmaNmr": [                     // optional
```

```
                {
                    "PnOffset": int,        // required
                    "CdmaChannel": int,     // required
                    "PilotPower": int,      // optional
                    "BaseStationId": int    // optional
                }
            ]
        }
    ]
  }
}
```

WCDMA (Wideband code-division multiple access)

When you use this measurement data, you must specify information such as the network and country code, signal power and identification information, the base station information, and optional additional parameters. The following code shows an example of the payload format. For more information about these parameters, see CDMA object.

```
{
    "Timestamp": 1664313161,           // optional
    "CellTowers": {
        "Wcdma": [
        {
          "Mcc": int,                   // required
          "Mnc": int,                   // required
          "UtranCid": int,              // required
          "Lac": int,                   // optional
          "WcdmaLocalId": {             // optional
              "Uarfcndl": int,          // required
              "Psc": int,               // required
          },
          "Rscp": int,                  // optional
          "Pathloss": int,              // optional
          "WcdmaNmr": [                 // optional
                {
                  "Uarfcndl": int,      // required
                  "Psc": int,           // required
                  "UtranCid": int,      // required
                  "Rscp": int,          // optional
                  "Pathloss": int,      // optional
                }
            ]
```

```
            }
         ]
     }
}
```

TD-SCDMA (Time division synchronous code-division multiple access)

When you use this measurement data, you must specify information such as the network and country code, signal power and identification information, the base station information, and optional additional parameters. The following code shows an example of the payload format. For more information about these parameters, see CDMA object.

```
{
    "Timestamp": 1664313161,           // optional
    "CellTowers": {
        "Tdscdma": [
        {
          "Mcc": int,                    // required
          "Mnc": int,                    // required
          "UtranCid": int,               // required
          "Lac": int,                    // optional
          "TdscdmaLocalId": {            // optional
              "Uarfcn": int,             // required
              "CellParams": int,         // required
          },
          "TdscdmaTimingAdvance": int,   // optional
          "Rscp": int,                   // optional
          "Pathloss": int,               // optional
          "TdscdmaNmr": [                // optional
                {
                  "Uarfcn": int,         // required
                  "CellParams": int,     // required
                  "UtranCid": int,       // optional
                  "Rscp": int,           // optional
                  "Pathloss": int,       // optional
                }
            ]
         }
      ]
   }
}
```

IP reverse lookup solver

You can use the IP reverse lookup solver to resolve the location using the IP address as input. The solver can obtain the location information from devices that have been provisioned with AWS IoT. Specify the IP address information using a format that's either the IPv4 or IPv6 standard pattern, or the IPv6 hex compressed pattern. You then obtain the resolved location estimate, including additional information such as city and country where the device is located.

> **Note**
>
> By using the IP reverse lookup, you agree not to use it for the purpose of identifying or locating a specific household or street address.

IP reverse lookup solver payload example

The following code shows an example of the JSON payload from the device that contains the measurement data. When AWS IoT Core Device Location receives the IP address information in the measurement data, it looks up this information in the solver provider's database, which is then used to resolve the location information. To retrieve the information, either provide the JSON payload using this format, or specify values for the Ip parameter of the GetPositionEstimate API operation.

> **Note**
>
> When this solver is used, the city, state, country, and postal code where the device is located is also reported in addition to the coordinates. For an example, see Resolving device location (console).

```
{
    "Timestamp": 1664313161,
    "Ip":{
        "IpAddress":"54.240.198.35"
      }
}
```

GNSS solver

Use the GNSS (Global Navigation Satellite System) solver to retrieve the device location using the information contained in the GNSS scan result messages or NAV messages. You can optionally provide additional GNSS assistance information, which reduces the number of variables that the solver must use to search for signals. By providing this assistance information, which includes the position, altitude, and the capture time and accuracy information, the solver can easily identify the satellites in view and compute the device location.

This solver can be used with LoRaWAN devices and Amazon Sidewalk enabled devices, as well as other devices that have been provisioned with AWS IoT devices that have been provisioned with AWS IoT. For general IoT devices, if the devices support location estimation using GNSS, when the GNSS scan information is received from the device, the transceivers resolve the location information. For LoRaWAN devices, the devices must have the LoRa Edge chipset. When an uplink message is received from the device, the GNSS scan data is sent to AWS IoT for LoRaWAN or AWS IoT for Amazon Sidewalk, and the location is estimated based on the scan results from the transceivers.

GNSS solver payload example

The following code shows an example of the JSON payload from the device that contains the measurement data. When AWS IoT Core Device Location receives the GNSS scan information containing the payload in the measurement data, it uses the transceivers and any additional assistance information included to search for signals and resolve the location information. To retrieve the information, either provide the JSON payload using this format, or specify values for the Gnss parameter of the GetPositionEstimate API operation.

> **Note**
>
> Before AWS IoT Core Device Location can resolve the device location, you must remove the destination byte from the payload.

```
{
    "Timestamp": 1664313161,                    // optional
    "Gnss": {
        "AssistAltitude": number,               // optional
        "AssistPosition": [ number ],           // optional
        "CaptureTime": number,                  // optional
```

```
        "CaptureTimeAccuracy": number,        // optional
        "Payload": "string",                  // required
        "Use2DSolver": boolean                // optional
    }
}
```

BLE solver for Amazon Sidewalk enabled devices

> **Note**
>
> For Bluetooth Low Energy based location, AWS IoT returns location coordinates based on the approximate location of nearby Sidewalk Gateways that are connected to Amazon Sidewalk and have the Community Finding feature enabled. Gateway Location Data is AWS Content and is provided to you solely for the purpose of assisting you in locating your devices that are connected to Amazon Sidewalk, and you must only use the data for that purpose. You must only use and access location data via the interface and functionality that we generally make available to you, and you must not attempt to re-identify, reverse engineer, or re-map any Gateway location data provided by us to you.

Amazon Sidewalk end devices connecting over AWS IoT Core for Amazon Sidewalk can resolve their location data using BLE, Wi-Fi, or GNSS uplink messages published by the Amazon Sidewalk end device. For more information, see the AWS IoT Core for Amazon Sidewalk developer guide.

Event messages

This section contains information about messages published by AWS IoT when things or jobs are updated or changed. For information about the AWS IoT Events service that allows you to create detectors to monitor your devices for failures or changes in operation, and to trigger actions when they occur, see AWS IoT Events.

How event messages are generated

AWS IoT publishes event messages when certain events occur. For example, events are generated by the registry when things are added, updated, or deleted. Each event causes a single event message to be sent. Event messages are published over MQTT with a JSON payload. The content of the payload depends on the type of event.

> **Note**
>
> Event messages are guaranteed to be published once. It is possible for them to be published more than once. The ordering of event messages is not guaranteed.

Policy for receiving event messages

To receive event messages, your device must use an appropriate policy that allows it to connect to the AWS IoT device gateway and subscribe to MQTT event topics. You must also subscribe to the appropriate topic filters.

The following is an example of the policy required for receiving lifecycle events:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe",
                "iot:Receive"
            ],
```

```
                "Resource": [
                    "arn:aws:iot:us-east-1:123456789012:/$aws/events/*"
                ]
            }
        ]
}
```

Enable events for AWS IoT

Before subscribers to the reserved topics can receive messages, you must enable event messages from the AWS Management Console or by using the API or CLI. For information about the event messages that the different options manage, see the Table of AWS IoT event configuration settings.

- To enable event messages, go to the Settings tab of the AWS IoT console and then, in the **Event-based messages** section, choose **Manage events**. You can specify the events that you want to manage.
- To control which event types are published by using the API or CLI, call the UpdateEventConfigurations API or use the **update-event-configurations** CLI command. For example:

```
aws iot update-event-configurations --event-configurations "{\"THING\":{\"Enabled\":
 true}}"
```

> **Note**
>
> All quotation marks (") are escaped with a backslash (\).

You can get the current event configuration by calling the DescribeEventConfigurations API or by using the **describe-event-configurations** CLI command. For example:.

```
aws iot describe-event-configurations
```

Table of AWS IoT event configuration settings

Event category (AWS IoT Console: Settings: Event-based messages)	`eventConfigurations` key value (AWS CLI/API)	Event message topic
(Can only be configured by using the AWS CLI/API)	`CA_CERTIFICATE`	`$aws/events/certificates/registered/ caCertificateId`
(Can only be configured by using the AWS CLI/API)	`CERTIFICATE`	`$aws/events/presence/connected/ clientId`
(Can only be configured by using the AWS CLI/API)	`CERTIFICATE`	`$aws/events/presence/disconnected/ clientId`
(Can only be configured by using the AWS CLI/API)	`CERTIFICATE`	`$aws/events/subscriptions/subscribed/ clientId`
(Can only be configured by using the AWS CLI/API)	`CERTIFICATE`	`$aws/events/subscriptions/unsubscribed/ clientId`
Job completed, canceled	`JOB`	`$aws/events/job/jobID/canceled`
Job completed, canceled	`JOB`	`$aws/events/job/jobID/cancellation_in_progress`
Job completed, canceled	`JOB`	`$aws/events/job/jobID/completed`
Job completed, canceled	`JOB`	`$aws/events/job/jobID/deleted`

Event category (AWS IoT Console: Settings: Event-based messages)	`eventConfigurations` key value (AWS CLI/API)	Event message topic
Job completed, canceled	`JOB`	`$aws/events/job/`*`jobID`*`/deletion_in_progress`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/canceled`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/deleted`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/failed`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/rejected`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/removed`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/succeeded`
Job execution: success, failed, rejected, canceled, removed	`JOB_EXECUTION`	`$aws/events/jobExecution/ `*`jobID`*`/timed_out`
Thing: created, updated, deleted	`THING`	`$aws/events/thing/ `*`thingName`*` /created`
Thing: created, updated, deleted	`THING`	`$aws/events/thing/ `*`thingName`*` /updated`

Event category (AWS IoT Console: Settings: Event-based messages)	`eventConfigurations` key value (AWS CLI/API)	Event message topic
Thing: created, updated, deleted	THING	`$aws/events/thing/` *`thingName`* `/deleted`
Thing group: added, removed	THING_GROUP	`$aws/events/thingGroup/` *`thingGroupName`* `/created`
Thing group: added, removed	THING_GROUP	`$aws/events/thingGroup/` *`thingGroupName`* `/updated`
Thing group: added, removed	THING_GROUP	`$aws/events/thingGroup/` *`thingGroupName`* `/deleted`
Thing group hierarchy: added, removed	THING_GROUP_HIERARCHY	`$aws/events/thingGroupHierarchy/thingGroup/` *`parentThingGroupName`* `/childThingGroup/` *`childThingGroupName`* `/added`
Thing group hierarchy: added, removed	THING_GROUP_HIERARCHY	`$aws/events/thingGroupHierarchy/thingGroup/` *`parentThingGroupName`* `/childThingGroup/` *`childThingGroupName`* `/removed`

Event category **(AWS IoT Console: Settings: Event-based messages)**	**`eventConfigurations` key value** **(AWS CLI/API)**	**Event message topic**
Thing group membership: added, removed	`THING_GROUP_MEMBERSHIP`	`$aws/events/thingGroupMembership/thingGroup/ *thingGroupName* /thing/*thingName* /added`
Thing group membership: added, removed	`THING_GROUP_MEMBERSHIP`	`$aws/events/thingGroupMembership/thingGroup/ *thingGroupName* /thing/*thingName* /removed`
Thing type: created, updated, deleted	`THING_TYPE`	`$aws/events/thingType/ *thingTypeName* / created`
Thing type: created, updated, deleted	`THING_TYPE`	`$aws/events/thingType/ *thingTypeName* / updated`
Thing type: created, updated, deleted	`THING_TYPE`	`$aws/events/thingType/ *thingTypeName* / deleted`

Event category (AWS IoT Console: Settings: Event-based messages)	`eventConfigurations` key value (AWS CLI/API)	Event message topic

Thing type association: added, removed	`THING_TYPE_ASSOCIATION`	`$aws/events/thingTypeAssociation/thing/ thingName /thingType/ thingTypeName /added`
		
		`$aws/events/thingTypeAssociation/thing/ thingName /thingType/ thingTypeName /removed`

Registry events

The registry can publish event messages when things, thing types, and thing groups are created, updated, or deleted. These events, however, are not available by default. For information about how to turn on these events, see Enable events for AWS IoT.

The registry can provide the following event types:

- Thing events
- Thing type events
- Thing group events

Thing events

Thing Created/Updated/Deleted

The registry publishes the following event messages when things are created, updated, or deleted:

- `$aws/events/thing/thingName/created`
- `$aws/events/thing/thingName/updated`

- `$aws/events/thing/thingName/deleted`

The messages contain the following example payload:


```
{
    "eventType" : "THING_EVENT",
    "eventId" : "f5ae9b94-8b8e-4d8e-8c8f-b3266dd89853",
    "timestamp" : 1234567890123,
    "operation" : "CREATED|UPDATED|DELETED",
    "accountId" : "123456789012",
    "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",
    "thingName" : "MyThing",
    "versionNumber" : 1,
    "thingTypeName" : null,
    "attributes": {
                "attribute3": "value3",
                "attribute1": "value1",
                "attribute2": "value2"
    }
}
```


The payloads contain the following attributes:

eventType

Set to "THING_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingId

The ID of the thing being created, updated, or deleted.

thingName

The name of the thing being created, updated, or deleted.

versionNumber

The version of the thing being created, updated, or deleted. This value is set to 1 when a thing is created. It is incremented by 1 each time the thing is updated.

thingTypeName

The thing type associated with the thing, if one exists. Otherwise, `null`.

attributes

A collection of name-value pairs associated with the thing.

Thing type events

Thing type related events:

- Thing Type Created/Updated/Deprecated/Undeprecated/Deleted
- Thing Type Associated or Disassociated with a Thing

Thing Type Created/Updated/Deprecated/Undeprecated/Deleted

The registry publishes the following event messages when thing types are created, updated, deprecated, undeprecated, or deleted:

- `$aws/events/thingType/thingTypeName/created`
- `$aws/events/thingType/thingTypeName/updated`
- `$aws/events/thingType/thingTypeName/deleted`

The message contains the following example payload:

```
{
    "eventType" : "THING_TYPE_EVENT",
    "eventId" : "8827376c-4b05-49a3-9b3b-733729df7ed5",
    "timestamp" : 1234567890123,
    "operation" : "CREATED|UPDATED|DELETED",
    "accountId" : "123456789012",
    "thingTypeId" : "c530ae83-32aa-4592-94d3-da29879d1aac",
    "thingTypeName" : "MyThingType",
    "isDeprecated" : false|true,
    "deprecationDate" : null,
    "searchableAttributes" : [ "attribute1", "attribute2", "attribute3" ],
    "propagatingAttributes": [
        {
     "userPropertyKey": "key",
     "thingAttribute": "model"
        },
        {
     "userPropertyKey": "key",
     "connectionAttribute": "iot:ClientId"
        }
    ],
    "description" : "My thing type"
}
```


The payloads contain the following attributes:

eventType

Set to "THING_TYPE_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingTypeId

The ID of the thing type being created, updated, deprecated, or deleted.

thingTypeName

The name of the thing type being created, updated, deprecated, or deleted.

isDeprecated

`true` if the thing type is deprecated. Otherwise, `false`.

deprecationDate

The UNIX timestamp for when the thing type was deprecated.

searchableAttributes

A collection of name-value pairs associated with the thing type that can be used for searching.

propagatingAttributes

A list of propagating attributes. A propagating attribute can contain a thing attribute, a connection attribute, and a user property key. For more information, see [Adding propagating attributes for message enrichment](#).

description

A description of the thing type.

Thing Type Associated or Disassociated with a Thing

The registry publishes the following event messages when a thing type is associated or disassociated with a thing.

- `$aws/events/thingTypeAssociation/thing/*thingName*/thingType/*typeName*/added`
- `$aws/events/thingTypeAssociation/thing/*thingName*/thingType/*typeName*/removed`

The following is an example of an `added` payload. Payloads for `removed` messages are similar.

```
{
    "eventId" : "87f8e095-531c-47b3-aab5-5171364d138d",
    "eventType" : "THING_TYPE_ASSOCIATION_EVENT",
    "operation" : "ADDED",
    "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",
    "thingName": "myThing",
    "thingTypeName" : "MyThingType",
    "timestamp" : 1234567890123,
}
```


The payloads contain the following attributes:

eventId

A unique event ID (string).

eventType

Set to "THING_TYPE_ASSOCIATION_EVENT".

operation

The operation that triggered the event. Valid values are:

- ADDED
- REMOVED

thingId

The ID of the thing whose type association was changed.

thingName

The name of the thing whose type association was changed.

thingTypeName

The thing type associated with, or no longer associated with, the thing.

timestamp

The UNIX timestamp of when the event occurred.

Thing group events

Thing group related events:

- Thing Group Created/Updated/Deleted
- Thing Added to or Removed from a Thing Group
- Thing Group Added to or Deleted from a Thing Group

Thing Group Created/Updated/Deleted

The registry publishes the following event messages when a thing group is created, updated, or deleted.

- `$aws/events/thingGroup/groupName/created`
- `$aws/events/thingGroup/groupName/updated`
- `$aws/events/thingGroup/groupName/deleted`

The following is an example of an `updated` payload. Payloads for `created` and `deleted` messages are similar.


```
{
  "eventType": "THING_GROUP_EVENT",
  "eventId": "8b9ea8626aeaa1e42100f3f32b975899",
  "timestamp": 1603995417409,
  "operation": "UPDATED",
  "accountId": "571EXAMPLE833",
  "thingGroupId": "8757eec8-bb37-4cca-a6fa-403b003d139f",
  "thingGroupName": "Tg_level5",
  "versionNumber": 3,
  "parentGroupName": "Tg_level4",
  "parentGroupId": "5fce366a-7875-4c0e-870b-79d8d1dce119",
  "description": "New description for Tg_level5",
  "rootToParentThingGroups": [
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/TgTopLevel",
      "groupId": "36aa0482-f80d-4e13-9bff-1c0a75c055f6"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level1",
      "groupId": "bc1643e1-5a85-4eac-b45a-92509cbe2a77"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level2",
      "groupId": "0476f3d2-9beb-48bb-ae2c-ea8bd6458158"
```

```
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level3",
      "groupId": "1d9d4ffe-a6b0-48d6-9de6-2e54d1eae78f"
    },
    {
      "groupArn": "arn:aws:iot:us-west-2:571EXAMPLE833:thinggroup/Tg_level4",
      "groupId": "5fce366a-7875-4c0e-870b-79d8d1dce119"
    }
  ],
  "attributes": {
    "attribute1": "value1",
    "attribute3": "value3",
    "attribute2": "value2"
  },
  "dynamicGroupMappingId": null
}
```


The payloads contain the following attributes:

eventType

Set to "THING_GROUP_EVENT".

eventId

A unique event ID (string).

timestamp

The UNIX timestamp of when the event occurred.

operation

The operation that triggered the event. Valid values are:

- CREATED
- UPDATED
- DELETED

accountId

Your AWS account ID.

thingGroupId

The ID of the thing group being created, updated, or deleted.

thingGroupName

The name of the thing group being created, updated, or deleted.

versionNumber

The version of the thing group. This value is set to 1 when a thing group is created. It is incremented by 1 each time the thing group is updated.

parentGroupName

The name of the parent thing group, if one exists.

parentGroupId

The ID of the parent thing group, if one exists.

description

A description of the thing group.

rootToParentThingGroups

An array of information about the parent thing group. There is one element for each parent thing group, starting from the root thing group and continuing to the thing group's parent. Each entry contains the thing group's `groupArn` and `groupId`.

attributes

A collection of name-value pairs associated with the thing group.

Thing Added to or Removed from a Thing Group

The registry publishes the following event messages when a thing is added to or removed from a thing group.

- `$aws/events/thingGroupMembership/thingGroup/`*`thingGroupName`*`/thing/`*`thingName`*`/added`
- `$aws/events/thingGroupMembership/thingGroup/`*`thingGroupName`*`/thing/`*`thingName`*`/removed`

The messages contain the following example payload:

```
{
```

```
    "eventType" : "THING_GROUP_MEMBERSHIP_EVENT",
    "eventId" : "d684bd5f-6f6e-48e1-950c-766ac7f02fd1",
    "timestamp" : 1234567890123,
    "operation" : "ADDED|REMOVED",
    "accountId" : "123456789012",
    "groupArn" : "arn:aws:iot:ap-northeast-2:123456789012:thinggroup/
MyChildThingGroup",
    "groupId" : "06838589-373f-4312-b1f2-53f2192291c4",
    "thingArn" : "arn:aws:iot:ap-northeast-2:123456789012:thing/MyThing",
    "thingId" : "b604f69c-aa9a-4d4a-829e-c480e958a0b5",
    "membershipId" : "8505ebf8-4d32-4286-80e9-c23a4a16bbd8"
}
```


The payloads contain the following attributes:

eventType

Set to "THING_GROUP_MEMBERSHIP_EVENT".

eventId

The event ID.

timestamp

The UNIX timestamp for when the event occurred.

operation

`ADDED` when a thing is added to a thing group. `REMOVED` when a thing is removed from a thing group.

accountId

Your AWS account ID.

groupArn

The ARN of the thing group.

groupId

The ID of the group.

thingArn

The ARN of the thing that was added or removed from the thing group.

thingId

The ID of the thing that was added or removed from the thing group.

membershipId

An ID that represents the relationship between the thing and the thing group. This value is generated when you add a thing to a thing group.

Thing Group Added to or Deleted from a Thing Group

The registry publishes the following event messages when a thing group is added to or removed from another thing group.

- `$aws/events/thingGroupHierarchy/thingGroup/parentThingGroupName/childThingGroup/childThingGroupName/added`
- `$aws/events/thingGroupHierarchy/thingGroup/parentThingGroupName/childThingGroup/childThingGroupName/removed`

The message contains the following example payload:

```
{
    "eventType" : "THING_GROUP_HIERARCHY_EVENT",
    "eventId" : "264192c7-b573-46ef-ab7b-489fcd47da41",
    "timestamp" : 1234567890123,
    "operation" : "ADDED|REMOVED",
    "accountId" : "123456789012",
    "thingGroupId" : "8f82a106-6b1d-4331-8984-a84db5f6f8cb",
    "thingGroupName" : "MyRootThingGroup",
    "childGroupId" : "06838589-373f-4312-b1f2-53f2192291c4",
    "childGroupName" : "MyChildThingGroup"
}
```

The payloads contain the following attributes:

eventType

Set to "THING_GROUP_HIERARCHY_EVENT".

eventId

The event ID.

timestamp

The UNIX timestamp for when the event occurred.

operation

`ADDED` when a thing is added to a thing group. `REMOVED` when a thing is removed from a thing group.

accountId

Your AWS account ID.

thingGroupId

The ID of the parent thing group.

thingGroupName

The name of the parent thing group.

childGroupId

The ID of the child thing group.

childGroupName

The name of the child thing group.

Jobs events

The AWS IoT Jobs service publishes to reserved topics on the MQTT protocol when jobs are pending, completed, or canceled, and when a device reports success or failure when running a job. Devices or management and monitoring applications can track the status of jobs by subscribing to these topics.

How to enable jobs events

Response messages from the AWS IoT Jobs service don't pass through the message broker and they can't be subscribed to by other clients or rules. To subscribe to job activity-related messages, use the `notify` and `notify-next` topics. For information about jobs topics, see Job topics.

To be notified of jobs updates, enable these jobs events by using the AWS Management Console, or by using the API or CLI. For more information, see Enable events for AWS IoT.

How jobs events work

Because it can take some time to cancel or delete a job, two messages are sent to indicate the start and end of a request. For example, when a cancellation request starts, a message is sent to the `$aws/events/job/jobID/cancellation_in_progress` topic. When the cancellation request is complete, a message is sent to the `$aws/events/job/jobID/canceled` topic.

A similar process occurs for a job deletion request. Management and monitoring applications can subscribe to these topics to keep track of the status of jobs. For more information about publishing and subscribing to MQTT topics, see [the section called “Device communication protocols”](#).

Job event types

The following examples show the different types of jobs events:

Job Completed/Canceled/Deleted

The AWS IoT Jobs service publishes a message on an MQTT topic when a job is completed, canceled, deleted, or when cancellation or deletion are in progress:

- `$aws/events/job/jobID/completed`
- `$aws/events/job/jobID/canceled`
- `$aws/events/job/jobID/deleted`
- `$aws/events/job/jobID/cancellation_in_progress`
- `$aws/events/job/jobID/deletion_in_progress`

The `completed` message contains the following example payload:


```
{
  "eventType": "JOB",
  "eventId": "7364ffd1-8b65-4824-85d5-6c14686c97c6",
  "timestamp": 1234567890,
  "operation": "completed",
  "jobId": "27450507-bf6f-4012-92af-bb8a1c8c4484",
  "status": "COMPLETED",
  "targetSelection": "SNAPSHOT|CONTINUOUS",
  "targets": [
    "arn:aws:iot:us-east-1:123456789012:thing/a39f6f91-70cf-4bd2-a381-9c66df1a80d0",
    "arn:aws:iot:us-east-1:123456789012:thinggroup/2fc4c0a4-6e45-4525-
a238-0fe8d3dd21bb"
```

```
  ],
  "description": "My Job Description",
  "completedAt": 1234567890123,
  "createdAt": 1234567890123,
  "lastUpdatedAt": 1234567890123,
  "jobProcessDetails": {
    "numberOfCanceledThings": 0,
    "numberOfRejectedThings": 0,
    "numberOfFailedThings": 0,
    "numberOfRemovedThings": 0,
    "numberOfSucceededThings": 3
  }
}
```


The `canceled` message contains the following example payload.


```
{
  "eventType": "JOB",
  "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
  "timestamp": 1234567890,
  "operation": "canceled",
  "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
  "status": "CANCELED",
  "targetSelection": "SNAPSHOT|CONTINUOUS",
  "targets": [
    "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
    "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
  ],
  "description": "My job description",
  "createdAt": 1234567890123,
  "lastUpdatedAt": 1234567890123
}
```


The `deleted` message contains the following example payload.


```
{
      "eventType": "JOB",
      "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
      "timestamp": 1234567890,
      "operation": "deleted",
      "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
```

```
      "status": "DELETED",
      "targetSelection": "SNAPSHOT|CONTINUOUS",
      "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
        "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
      ],
      "description": "My job description",
      "createdAt": 1234567890123,
      "lastUpdatedAt": 1234567890123,
      "comment": "Comment for this operation"
    }
```


The `cancellation_in_progress` message contains the following example payload:


```
{
      "eventType": "JOB",
      "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
      "timestamp": 1234567890,
      "operation": "cancellation_in_progress",
      "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
      "status": "CANCELLATION_IN_PROGRESS",
      "targetSelection": "SNAPSHOT|CONTINUOUS",
      "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
        "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
      ],
      "description": "My job description",
      "createdAt": 1234567890123,
      "lastUpdatedAt": 1234567890123,
      "comment": "Comment for this operation"
    }
```


The `deletion_in_progress` message contains the following example payload:


```
{
      "eventType": "JOB",
      "eventId": "568d2ade-2e9c-46e6-a115-18afa1286b06",
      "timestamp": 1234567890,
      "operation": "deletion_in_progress",
```

```
      "jobId": "4d2a531a-da2e-47bb-8b9e-ff5adcd53ef0",
      "status": "DELETION_IN_PROGRESS",
      "targetSelection": "SNAPSHOT|CONTINUOUS",
      "targets": [
        "arn:aws:iot:us-east-1:123456789012:thing/Thing0-947b9c0c-ff10-4a80-b4b3-
cd33d0145a0f",
        "arn:aws:iot:us-east-1:123456789012:thinggroup/
ThingGroup1-95c644d5-1621-41a6-9aa5-ad2de581d18f"
      ],
      "description": "My job description",
      "createdAt": 1234567890123,
      "lastUpdatedAt": 1234567890123,
      "comment": "Comment for this operation"
    }
```


Job Execution Terminal Status

The AWS IoT Jobs service publishes a message when a device updates a job execution to terminal status:

- `$aws/events/jobExecution/jobID/succeeded`
- `$aws/events/jobExecution/jobID/failed`
- `$aws/events/jobExecution/jobID/rejected`
- `$aws/events/jobExecution/jobID/canceled`
- `$aws/events/jobExecution/jobID/timed_out`
- `$aws/events/jobExecution/jobID/removed`
- `$aws/events/jobExecution/jobID/deleted`

The message contains the following example payload:


```
{
  "eventType": "JOB_EXECUTION",
  "eventId": "cca89fa5-8a7f-4ced-8c20-5e653afb3572",
  "timestamp": 1234567890,
  "operation": "succeeded|failed|rejected|canceled|removed|timed_out",
  "jobId": "154b39e5-60b0-48a4-9b73-f6f8dd032d27",
  "thingArn": "arn:aws:iot:us-east-1:123456789012:myThing/6d639fbc-8f85-4a90-924d-
a2867f8366a7",
  "status": "SUCCEEDED|FAILED|REJECTED|CANCELED|REMOVED|TIMED_OUT",
  "statusDetails": {
```

```
    "key": "value"
  }
}
```

Lifecycle events

AWS IoT can publish lifecycle events on the MQTT topics. These events are available by default and they can't be disabled.

> **Note**
>
> Lifecycle messages might be sent out of order. You might receive duplicate messages. `thingName` will only be included if the client is connecting using the exclusive thing feature.

In this topic:

- Connect/Disconnect events
- Connect attempt failure event
- Subscribe/Unsubscribe events

Connect/Disconnect events

> **Note**
>
> With AWS IoT Device Management fleet indexing, you can search for things, run aggregate queries, and create dynamic groups based on thing Connect/Disconnect events. For more information, see Fleet indexing.

AWS IoT publishes a message to the following MQTT topics when a client connects or disconnects:

- `$aws/events/presence/connected/clientId` – A client connected to the message broker.
- `$aws/events/presence/disconnected/clientId` – A client disconnected from the message broker.

The following is a list of JSON elements that are contained in the connection/disconnection messages published to the `$aws/events/presence/connected/clientId` topic.

clientId

The client ID of the connecting or disconnecting client.

> **Note**
>
> Client IDs that contain # or + do not receive lifecycle events.

thingName

The name of your IoT thing. `thingName` will only be included if the client is connecting using the exclusive thing feature.

clientInitiatedDisconnect

True if the client initiated the disconnect. Otherwise, false. Found in disconnect messages only.

disconnectReason

The reason why the client is disconnecting. Found in disconnect messages only. The following table contains valid values and whether the broker will send Last Will and Testament (LWT) messages when the disconnection occurs.

Disconnect reason	Description	The broker will send the LWT messages
`AUTH_ERROR`	The client failed to authenticate or authorization failed.	Yes[1]
`CLIENT_INITIATED_DISCONNECT`	The client indicates that it will disconnect. The client can do this by sending either a MQTT `DISCONNECT` control packet or a `Close frame` if the client is using a WebSocket connection.	No
`CLIENT_ERROR`	The client did something wrong that causes it to disconnect. For example, a client will be	Yes

Disconnect reason	Description	The broker will send the LWT messages
	disconnected for sending more than 1 MQTT `CONNECT` packet on the same connection or if the client attempts to publish with a payload that exceeds the payload limit.	
`CONNECTION_LOST`	The client-server connection is cut off. This can happen during a period of high network latency or when the internet connection is lost.	Yes
`DUPLICATE_CLIENTID`	The client is using a client ID that is already in use. In this case, the client that is already connected will be disconnected with this disconnect reason.	Yes
`FORBIDDEN_ACCESS`	The client is not allowed to be connected. For example, a client with a denied IP address will fail to connect.	Yes[1]
`MQTT_KEEP_ALIVE_TIMEOUT`	If there is no client-server communication for 1.5x of the client's keep-alive time, the client is disconnected.	Yes
`SERVER_ERROR`	Disconnected due to unexpected server issues.	Yes
`SERVER_INITIATED_DISCONNECT`	Server intentionally disconnects a client for operational reasons.	Yes
`API_INITIATED_DISCONNECT`	The client was disconnected using the `DeleteConnection` API.	Yes[2]
`THROTTLED`	The client is disconnected for exceeding a throttling limit.	Yes

Disconnect reason	Description	The broker will send the LWT messages
WEBSOCKET_TTL_EXPIRATION	The client is disconnected because a WebSocket has been connected longer than its time-to-live value.	Yes
CUSTOMAUTH_TTL_EXPIRATION	The client is disconnected because it has been connected longer than the time-to-live value of its custom authorizer.	Yes

[1]If the device has an active connection before receiving this error.

[2]To prevent Last Will and Testament (LWT) messages, set `preventWillMessage=true` to override the `DeleteConnection` API's default LWT sending behavior.

eventType

The type of event. Valid values are `connected` or `disconnected`.

ipAddress

The IP address of the connecting client. This can be in IPv4 or IPv6 format. Found in connection messages only.

principalIdentifier

The credential used to authenticate. For TLS mutual authentication certificates, this is the certificate ID. For other connections, this is IAM credentials.

sessionIdentifier

A globally unique identifier in AWS IoT that exists for the life of the session.

timestamp

An approximation of when the event occurred.

versionNumber

The version number for the lifecycle event. This is a monotonically increasing long integer value for each client ID connection. The version number can be used by a subscriber to infer the order of lifecycle events.

> **Note**
>
> The connect and disconnect messages for a client connection have the same version number.
> The version number might skip values and is not guaranteed to be consistently increasing by 1 for each event.
> If a client is not connected for approximately one hour, the version number is reset to 0. For persistent sessions, the version number is reset to 0 after a client has been disconnected longer than the configured time-to-live (TTL) for the persistent session.

A connect message has the following structure.

```
{
    "clientId": "186b5",
    "thingName": "exampleThing",
    "timestamp": 1573002230757,
    "eventType": "connected",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "12345678901234567890123456789012",
    "ipAddress": "192.0.2.0",
    "versionNumber": 0
}
```

A disconnect message has the following structure.

```
{
    "clientId": "186b5",
    "thingName": "exampleThing",
    "timestamp": 1573002340451,
    "eventType": "disconnected",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "12345678901234567890123456789012",
    "clientInitiatedDisconnect": true,
    "disconnectReason": "CLIENT_INITIATED_DISCONNECT",
    "versionNumber": 0
}
```

Handling client disconnections

The best practice is to always have a wait state implemented for lifecycle events, including Last Will and Testament (LWT) messages. When a disconnect message is received, your code should wait a period of time and verify a device is still offline before taking action. One way to do this is by using SQS Delay Queues. When a client receives a LWT or a lifecycle event, you can enqueue a message (for example, for 5 seconds). When that message becomes available and is processed (by Lambda or another service), you can first check if the device is still offline before taking further action.

Connect attempt failure event

AWS IoT publishes a message to the following MQTT topic when a client is not authorized to connect or when a last will and testament is configured and the client is not authorized to publish to that last will topic.

```
$aws/events/presence/connect_failed/clientId
```

The following is a list of JSON elements that are contained in the connect authorization messages published to the `$aws/events/presence/connect_failed/clientId` topic.

clientId

The client ID of the client that attempted and failed to connect.

> **Note**
>
> Client IDs that contain # or + do not receive lifecycle events.

thingName

The name of your IoT thing. `thingName` will only be included if the client is connecting using the exclusive thing feature.

timestamp

An approximation of when the event occurred.

eventType

The type of event. Valid value is `connect_failed`.

connectFailureReason

The reason why the connection fails. Valid value is AUTHORIZATION_FAILED.

principalIdentifier

The credential used to authenticate. For TLS mutual authentication certificates, this is the certificate ID. For other connections, this is IAM credentials.

sessionIdentifier

A globally unique identifier in AWS IoT that exists for the life of the session.

ipAddress

The IP address of the connecting client. This can be in IPv4 or IPv6 format. Found in connection messages only.

A connection failure message has the following structure.

```
{
    "clientId": "186b5",
    "thingName": "exampleThing",
    "timestamp": 1460065214626,
    "eventType": "connect_failed",
    "connectFailureReason": "AUTHORIZATION_FAILED",
    "principalIdentifier": "12345678901234567890123456789012",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "ipAddress" : "192.0.2.0"
}
```

Subscribe/Unsubscribe events

AWS IoT publishes a message to the following MQTT topic when a client subscribes or unsubscribes to an MQTT topic:

```
$aws/events/subscriptions/subscribed/clientId
```

or

```
$aws/events/subscriptions/unsubscribed/clientId
```

Where `clientId` is the MQTT client ID that connects to the AWS IoT message broker.

The message published to this topic has the following structure:


```
{
    "clientId": "186b5",
    "thingName": "exampleThing",
    "timestamp": 1460065214626,
    "eventType": "subscribed" | "unsubscribed",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "12345678901234567890123456789012",
    "topics" : ["foo/bar","device/data","dog/cat"]
}
```


The following is a list of JSON elements that are contained in the subscribed and unsubscribed messages published to the `$aws/events/subscriptions/subscribed/clientId` and `$aws/events/subscriptions/unsubscribed/clientId` topics.

clientId

The client ID of the subscribing or unsubscribing client.

> **Note**
>
> Client IDs that contain # or + do not receive lifecycle events.

thingName

The name of your IoT thing. `thingName` will only be included if the client is connecting using the exclusive thing feature.

eventType

The type of event. Valid values are `subscribed` or `unsubscribed`.

principalIdentifier

The credential used to authenticate. For TLS mutual authentication certificates, this is the certificate ID. For other connections, this is IAM credentials.

sessionIdentifier

A globally unique identifier in AWS IoT that exists for the life of the session.

timestamp

An approximation of when the event occurred.

topics

An array of the MQTT topics to which the client has subscribed.

Note

Lifecycle messages might be sent out of order. You might receive duplicate messages.

Troubleshooting AWS IoT

Help us improve this topic

Let us know what would help make it better

The following information might help you troubleshoot common issues in AWS IoT.

Tasks

- AWS IoT Core troubleshooting guide
- AWS IoT Device Management troubleshooting guide
- AWS IoT Device Advisor troubleshooting guide
- AWS IoT errors

AWS IoT Core troubleshooting guide

Help us improve this topic

Let us know what would help make it better

This is the troubleshooting section for AWS IoT Core.

Topics

- Diagnosing connectivity issues
- Diagnosing rules issues
- Diagnosing problems with shadows
- Diagnosing Salesforce IoT input stream action issues
- Diagnosing Stream Limits
- Troubleshooting device fleet disconnects

Diagnosing connectivity issues

Help us improve this topic

Let us know what would help make it better

A successful connection to AWS IoT requires:

- A valid connection
- A valid and active certificate
- A policy that allows the desired connection and operation

Connection

How do I find the correct endpoint?

- The `endpointAddress` returned by `aws iot describe-endpoint --endpoint-type iot:Data-ATS`

  or
- The `domainName` returned by `aws iot describe-domain-configuration --domain-configuration-name "domain_configuration_name"`

How do I find the correct Server Name Indication (SNI) value?

The correct SNI value is the `endpointAddress` returned by the **describe-endpoint** or the `domainName` returned by the **describe-domain-configuration** commands. It's the same address as the endpoint in the previous step. When connecting devices to AWS IoT Core, clients can send the Server Name Indication (SNI) extension, which is not required but highly recommended. To use features such as multi-account registration, custom domains, and VPC endpoints, you must use the SNI extension. For more information, see Transport Security in AWS IoT.

How do I solve a connectivity issue that persists?

You can use AWS Device Advisor to help troubleshoot. Device Advisor's pre-built tests help you validate your device software against best practices for usage of TLS, MQTT, AWS IoT Device Shadow, and AWS IoT Jobs.

Here is a link to the existing Device Advisor content.

Authentication

Devices must be authenticated to connect to AWS IoT endpoints. For devices that use X.509 client certificates for authentication, the certificates must be registered with AWS IoT and be active.

How do my devices authenticate AWS IoT endpoints?

Add the AWS IoT CA certificate to your client's trust store. Refer to the documentation on Server Authentication in AWS IoT Core and then follow the links to download the appropriate CA certificate.

What is checked when a device connects to AWS IoT?

When a device attempts to connect to AWS IoT:

1. AWS IoT checks for a valid certificate and Server Name Indication (SNI) value.
2. AWS IoT checks to see that the certificate used is registered with the AWS IoT Account and that it has been activated.
3. When a device attempts to perform any action in AWS IoT, such as to subscribe to or publish a message, the policy attached to the certificate it used to connect is checked to confirm that the device is authorized to perform that action.

How can I validate a correctly configured certificate?

Use the OpenSSL `s_client` command to test a connection to the AWS IoT endpoint:

```
openssl s_client -connect custom_endpoint.iot.aws-region.amazonaws.com:8443 -
CAfile CA.pem -cert cert.pem -key privateKey.pem
```

For more information about using `openssl s_client`, see OpenSSL s_client documentation.

How do I check the status of a certificate?

- **List the certificates**

  If you don't know the certificate ID, you can see the status of all your certificates by using the `aws iot list-certificates` command.

- **Show a certificate's details**

  If you know the certificate's ID, this command shows you more detailed information about the certificate.

```
aws iot describe-certificate --certificate-id "certificateId"
```

- **Review the certificate in the AWS IoT Console**

  In the AWS IoT console, in the left menu, choose **Secure**, and then choose **Certificates**.

  Choose the certificate that you are using to connect from the list to open its detail page.

  In the certificate's detail page, you can see its current status.

  The certificate's status can be changed by using the **Actions** menu in the upper-right corner of the details page.

Authorization

AWS IoT resources use AWS IoT Core policies to authorize those resources to perform actions. For an action to be authorized, the specified AWS IoT resources must have a policy document attached to it that grants permission to perform that action.

I received a PUBNACK or SUBNACK response from the broker. What do I do?

Make sure that there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Make sure the attached policy authorizes the actions you are trying to perform.

Make sure the attached policy authorizes the resources that are trying to perform the authorized actions.

I have an *AUTHORIZATION_FAILURE* entry in my logs.

Make sure that there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Make sure the attached policy authorizes the actions you are trying to perform.

Make sure the attached policy authorizes the resources that are trying to perform the authorized actions.

How do I check what the policy authorizes?

In the AWS IoT console, in the left menu, choose **Security**, and then choose **Certificates**.

Choose the certificate that you are using to connect from the list to open its detail page.

In the certificate's detail page, you can see its current status.

In the left menu of the certificate's detail page, choose **Policies** to see the policies attached to the certificate.

Choose the desired policy to see its details page.

In the policy's details page, review the policy's **Policy document** to see what it authorizes.

Choose **Edit policy document** to make changes to the policy document.

Security and identity

When you provide the server certificates for AWS IoT custom domain configuration, the certificates have a maximum of four domain names.

For more information, see AWS IoT Core endpoints and quotas.

Diagnosing rules issues

Help us improve this topic

Let us know what would help make it better

This section describes some of the things to check when you encounter a problem with rule.

Configuring CloudWatch Logs for troubleshooting

The best way to debug issues you are having with rules is to use CloudWatch Logs. When you enable CloudWatch Logs for AWS IoT, you can see which rules are triggered and their success or failure. You also get information about whether WHERE clause conditions match. For more information, see Monitor AWS IoT using CloudWatch Logs.

The most common rules issue is authorization. The logs show if your role is not authorized to perform AssumeRole on the resource. Here is an example log generated by fine-grained logging:

```
{
```

```
    "timestamp": "2017-12-09 22:49:17.954",
    "logLevel": "ERROR",
    "traceId": "ff563525-6469-506a-e141-78d40375fc4e",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "RuleExecution",
    "clientId": "iotconsole-123456789012-3",
    "topicName": "test-topic",
    "ruleName": "rule1",
    "ruleAction": "DynamoAction",
    "resources": {
        "ItemHashKeyField": "id",
        "Table": "trashbin",
        "Operation": "Insert",
        "ItemHashKeyValue": "id",
        "IsPayloadJSON": "true"
    },
    "principalId": "ABCDEFG1234567ABCD890:outis",
    "details": "User: arn:aws:sts::123456789012:assumed-role/dynamo-
testbin/5aUMInJH is not authorized to perform: dynamodb:PutItem on
 resource: arn:aws:dynamodb:us-east-1:123456789012:table/testbin (Service:
 AmazonDynamoDBv2; Status Code: 400; Error Code: AccessDeniedException; Request ID:
 AKQJ987654321AKQJ123456789AKQJ987654321AKQJ987654321)"
}
```


Here is a similar example log generated by global logging:

```
2017-12-09 22:49:17.954 TRACEID:ff562535-6964-506a-e141-78d40375fc4e
PRINCIPALID:ABCDEFG1234567ABCD890:outis [ERROR] EVENT:DynamoActionFailure
TOPICNAME:test-topic CLIENTID:iotconsole-123456789012-3
MESSAGE:Dynamo Insert record failed. The error received was User:
 arn:aws:sts::123456789012:assumed-role/dynamo-testbin/5aUMInJI is not authorized to
 perform: dynamodb:PutItem on resource: arn:aws:dynamodb:us-east-1:123456789012:table/
testbin
(Service: AmazonDynamoDBv2; Status Code: 400; Error Code: AccessDeniedException;
 Request ID: AKQJ987654321AKQJ987654321AKQJ987654321AKQJ987654321).
Message arrived on: test-topic, Action: dynamo, Table: trashbin, HashKeyField: id,
 HashKeyValue: id, RangeKeyField: None, RangeKeyValue: 123456789012
No newer events found at the moment. Retry.
```

For more information, see the section called “Viewing AWS IoT logs in the CloudWatch console”.

Diagnosing external services

External services are controlled by the end user. Before rule execution, make sure that the external services you have linked to your rule are set up and have enough throughput and capacity units for your application.

Diagnosing SQL problems

If your SQL query is not returning the data you expect:

- **Review the logs for error messages.**
- **Confirm that your SQL syntax matches the JSON document in the message.**

  Review the object and property names used in the query with those used in the JSON document of the topic's message payload. For more information about the JSON formatting in SQL queries, see JSON extensions.
- **Check to see if the JSON object or property names include reserved or numeric characters.**

  For more information about reserved characters in JSON object references in SQL queries, see JSON extensions.

Diagnosing problems with shadows

Help us improve this topic

Let us know what would help make it better

Diagnosing shadows

Issue	Troubleshooting guidelines
A device's shadow document is rejected with `Invalid JSON document.`	If you are unfamiliar with JSON, modify the examples provided in this guide for your own use. For more information, see Shadow document examples.

Issue	Troubleshooting guidelines
I submitted correct JSON, but none or only parts of it are stored in the device's shadow document.	Be sure you are following the JSON formatting guidelines. Only JSON fields in the `desired` and `reported` sections are stored. JSON content (even if formally correct) outside of those sections is ignored.
I received an error that the device's shadow exceeds the allowed size.	The device's shadow supports 8 KB of data only. Try shortening field names inside of your JSON document or simply create more shadows by creating more things. A device can have an unlimited number of things/shadows associated with it. The only requirement is that each thing name must be unique in your account.
When I receive a device's shadow, it is larger than 8 KB. How can this happen?	Upon receipt, the AWS IoT service adds metadata to the device's shadow. The service includes this data in its response, but it does not count toward the limit of 8 KB. Only the data for `desired` and `reported` state inside the state document sent to the device's shadow counts toward the limit.
My request has been rejected due to incorrect version. What should I do?	Perform a GET operation to sync to the latest state document version. When using MQTT, subscribe to the ./update/accepted topic to be notified about state changes and receive the latest version of the JSON document.

Issue	Troubleshooting guidelines
The timestamp is off by several seconds.	The timestamp for individual fields and the whole JSON document is updated when the document is received by the AWS IoT service or when the state document is published onto the ./update/accepted and ./update/delta message. Messages can be delayed over the network, which can cause the timestamp to be off by a few seconds.
My device can publish and subscribe on the corresponding shadow topics, but when I attempt to update the shadow document over the HTTP REST API, I get HTTP 403.	Be sure you have created policies in IAM to allow access to these topics and for the corresponding action (UPDATE/GET/DELETE) for the credentials you are using. IAM policies and certificate policies are independent.
Other issues.	The Device Shadow service logs errors to CloudWatch Logs. To identify device and configuration issues, enable CloudWatch Logs and view the logs for debug information.

Diagnosing Salesforce IoT input stream action issues

Help us improve this topic

Let us know what would help make it better

Execution trace

How do I see the execution trace of a Salesforce action?

See the Monitor AWS IoT using CloudWatch Logs section. After you have activated the logs, you can see the execution trace of the Salesforce action.

Action success and failure

How do I check that messages have been sent successfully to a Salesforce IoT input stream?

View the logs generated by execution of the Salesforce action in CloudWatch Logs. If you see `Action executed successfully`, then it means that the AWS IoT rules engine received confirmation from the Salesforce IoT that the message was successfully pushed to the targeted input stream.

If you are experiencing problems with the Salesforce IoT platform, contact Salesforce IoT support.

What do I do if messages have not been sent successfully to a Salesforce IoT input stream?

View the logs generated by execution of the Salesforce action in CloudWatch Logs. Depending on the log entry, you can try the following actions:

`Failed to locate the host`

Check that the `url` parameter of the action is correct and that your Salesforce IoT input stream exists.

`Received Internal Server Error from Salesforce`

Retry. If the problem persists, contact Salesforce IoT Support.

`Received Bad Request Exception from Salesforce`

Check the payload you are sending for errors.

`Received Unsupported Media Type Exception from Salesforce`

Salesforce IoT does not support a binary payload at this time. Check that you are sending a JSON payload.

`Received Unauthorized Exception from Salesforce`

Check that the `token` parameter of the action is correct and that your token is still valid.

`Received Not Found Exception from Salesforce`

Check that the `url` parameter of the action is correct and that your Salesforce IoT input stream exists.

If you receive an error that is not listed here, contact AWS IoT Support.

Diagnosing Stream Limits

Troubleshooting "Stream limit exceeded for your AWS account"

If you see "`Error: You have exceeded the limit for the number of streams in your AWS account.`", you can clean up the unused streams in your account instead of requesting a limit increase.

To clean up an unused stream that you created using the AWS CLI or SDK:

```
aws iot delete-stream –stream-id value
```

For more details, see delete-stream.

> **Note**
>
> You can use the `list-streams` command to find the stream IDs.

Troubleshooting device fleet disconnects

> **Help us improve this topic**
>
> Let us know what would help make it better

AWS IoT device fleet disconnects can happen for multiple reasons. This article explains how to diagnose a disconnect reason and how to handle disconnects caused by regular maintenance of AWS IoT service or a throttling limit.

To diagnose the disconnect reason

You can check the AWSIotLogsV2 log group in CloudWatch to identify the disconnect reason in the `disconnectReason` field of the log entry.

You can also use AWS IoT's lifecycle events feature to identify the disconnect reason. If you've subscribed to lifecycle's disconnect event (`$aws/events/presence/disconnected/clientId`), you'll get a notification from AWS IoT when the disconnect happens. You can identify the disconnect reason in the `disconnectReason` field of the notification.

For more information, see CloudWatch AWS IoT log entries and Lifecycle events.

To troubleshoot disconnects due to AWS IoT service maintenance

Disconnects caused by AWS IoT's service maintenance are logged as `SERVER_INITIATED_DISCONNECT` in AWS IoT's lifecycle event and CloudWatch. To handle these disconnects, adjust your client-side setup to make sure your devices can be automatically reconnected to the AWS IoT platform.

To troubleshoot disconnects due to a throttling limit

Disconnects caused by a throttling limit are logged as `THROTTLED` in AWS IoT's lifecycle event and CloudWatch. To handle these disconnects, you can request message broker limit increases as the device count grows.

For more information, see AWS IoT Core Message Broker.

AWS IoT Device Management troubleshooting guide

> **Help us improve this topic**
>
> Let us know what would help make it better

This is the troubleshooting section for AWS IoT Device Management.

Topics

- AWS IoT Jobs Troubleshooting
- Fleet Indexing Troubleshooting
- AWS IoT Device Management Software Package Catalog Troubleshooting

AWS IoT Jobs Troubleshooting

This is the troubleshooting section for AWS IoT Jobs.

How do I locate an AWS IoT Jobs endpoint?

How do I locate the AWS IoT Jobs control plane endpoint?

AWS IoT Jobs supports controls plane API operations using the HTTPS protocol. Verify you have connected to the correct control plane endpoint using the HTTPS protocol.

For a list of AWS region-specific endpoints, see AWS IoT Core - control plane endpoints.

For a list of FIPS compliant **AWS IoT Jobs control plane** endpoints, see FIPS Endpoints by Service

> **Note**
>
> AWS IoT Jobs and AWS IoT Core share the same AWS Region-specific endpoints.

How do I locate the AWS IoT Jobs data plane endpoint?

AWS IoT Jobs supports data plane API operations using the HTTPS and MQTT protocols. Verify you have connected to the correct data plane endpoint using the HTTPS or MQTT protocol.

- HTTPS protocol
  - Use the following **describe-endpoint** CLI command shown below or the `DescribeEndpoint` REST API. For the endpoint type, use `iot:Jobs`.

    ```
    aws iot describe-endpoint --endpoint-type iot:Jobs
    ```
- MQTT protocol
  - Use the following **describe-endpoint** CLI command shown below or the `DescribeEndpoint` REST API. For the endpoint type, use `iot:Data-ATS`.

    ```
    aws iot describe-endpoint --endpoint-type iot:Data-ATS
    ```

For a list of FIPS compliant **AWS IoT Jobs data plane** endpoints, see FIPS Endpoints by Service

How do I monitor AWS IoT Jobs activity and provide metrics?

Monitoring AWS IoT Jobs activity using Amazon CloudWatch provides real-time visibility into ongoing AWS IoT Jobs operations and helps control costs with CloudWatch alarms via AWS

IoT Rules. You must configure logging before you can monitor AWS IoT Jobs activity and setup CloudWatch alarms. For more information on setting up logging, see Configure AWS IoT logging.

For more information on Amazon CloudWatch and how to setup permission via an IAM user role to use CloudWatch resources, see Identity and access management for Amazon CloudWatch.

How do I set up AWS IoT Jobs metrics and monitoring using Amazon CloudWatch?

To set up AWS IoT logging, follow the steps outlined in Configure AWS IoT logging. AWS IoT logging set up can be done in the AWS Management Console, AWS CLI, or API. AWS IoT logging set up for specific thing groups must be done in the AWS CLI or API only.

The AWS IoT Jobs metrics section contains the AWS IoT Jobs metrics used for monitoring AWS IoT Jobs activity. It explains how to view the metrics in the AWS Management Console and AWS CLI.

Additionally, you can set up CloudWatch alarms to alert you of specific metrics you want to closely monitor. For guidance on alarm setup, see Using Amazon CloudWatch alarms.

Device fleets and single device troubleshooting

A job execution maintains a status of `QUEUED` indefinitely

When a job execution with a status state of `QUEUED` does not proceed to the next logical status state such as `IN_PROGRESS`, `FAILED`, or `TIMED_OUT`, one of the following scenarios may be the cause:

- Review your device activity in the CloudWatch logs located in the CloudWatch console. For more information, refer to Monitor AWS IoT using CloudWatch Logs.
- The IAM role associated with the job and subsequent job execution may not have the correct permissions listed in one of the policy statements of the IAM policy attached to that IAM role. Use the `describe-job` API to identify the IAM role linked to that job and subsequent job execution and review the IAM policy for correct permissions. Once the policy permission statements have been updated, you should be able to perform the `AssumeRole` API command on the resource.

A job execution was not created for my thing or thing group

When a job updates its status state to `IN_PROGRESS`, it will begin the job document rollout to all devices in your target group. This status state update will create a job execution for each target device. If a job execution was not created for one of the target devices, refer to the following guidance:

- Is the `thing` *directly* targeted by the job, the job has a status state of `IN_PROGRESS`, and the job is concurrent? If all three conditions are met, then the job is still sending out job executions to all devices in your target group and that specific `thing` has not received its job execution yet.
  - Review the devices in your target group for the job and the job status state in the AWS Management Console or use the [`describe-job`](#) API command.
  - Use the [`describe-job`](#) API command to review if the job has the `IsConcurrent` property set to true or false. For more information, see [Job limits](#).
- The `thing` is *not directly* targeted by the job.
  - If the `Thing` was added to a `ThingGroup` and the job targeted the `ThingGroup`, then verify the `Thing` is part of the `ThingGroup`.
  - If the job is a snapshot job with a status state of `IN_PROGRESS` and is concurrent, then the job is still sending out job executions to all devices in your target group and that specific `Thing` has not received its job execution yet.
  - If the job is a continuous job with a status state of `IN_PROGRESS` and is concurrent, then the job is still sending out job executions to all devices in your target group and that specific `Thing` has not received its job execution yet. For continuous jobs only, you can also remove the `Thing` from the `ThingGroup` and then add the `Thing` back to the `ThingGroup`.
  - If the job is a snapshot job with a status state of `IN_PROGRESS` and is not concurrent, then it's likely the `Thing` or `ThingGroup` membership relationship is not acknowledged by AWS IoT Jobs. It is recommended to add several seconds of waiting time after your `AddThingToThingGroup` call before you create your `Job`. Alternatively, you can switch the target selection to `Continuous`, thus making the service backfill the delayed `Thing` and `ThingGroup` membership attachment event.

New job fails due to `LimitedExceededException` error

If your job creation fails with an error response of `LimitedExceededException`, then call the `list-jobs` API and review all jobs with `isConcurrent=true` to determine if you are at your job concurrency limit. See [Job limits](#) for additional information on concurrent jobs. To view your job concurrency limits and to request a limit increase, see [AWS IoT Device Management jobs limits and quotas](#).

Job document size limit

The job document size is limited by the MQTT payload size. If you need a job document larger than 32 kB (kilobytes), 32,000 B (bytes), then create and store the job document in Amazon S3 and add an Amazon S3 object URL in the `documentSource` field for the `CreateJob` API or using the AWS CLI. For the AWS Management Console, add an Amazon S3 object URL in the Amazon S3 URL text box when creating a job.

- AWS Management Console create job documentation: Create and manage jobs by using the AWS Management Console
- AWS CLI create job documentation: Create and manage jobs using the AWS CLI
- `CreateJob` API documentation: CreateJob

Device Side MQTT message requests throttle limits

If you receive an error code 400 `ThrottlingException`, the device side MQTT message failed due to reaching the limit of simultaneous device side requests. See AWS IoT Device Management jobs limits and quotas for more information on throttle limits and if it is adjustable.

Connection timeout error

An error code 400 `RequestExpired` indicates a connection failure due to high latency or low client side timeout values.

- See Testing connectivity with your device data endpoint for information on testing connection between the client side and server side.

Invalid API command

Confirm the correct API command is entered to avoid an error message stating the API command is invalid. See the AWS IoT API Reference for a comprehensive list of all AWS IoT API commands.

Service side connection error

An error code 503 `ServiceUnavailable` indicates the error originated from the server side.

- See AWS Health Dashboard (all AWS services) for the current status of all AWS services.

- See [AWS Health Dashboard (personal AWS account)] for the current status of your personal AWS account.

Fleet Indexing Troubleshooting

Troubleshooting aggregation queries for the fleet indexing service

If you are having type mismatch errors, you can use CloudWatch Logs to troubleshoot the problem. CloudWatch Logs must be enabled before logs are written by the Fleet Indexing service. For more information, see Monitor AWS IoT using CloudWatch Logs.

To make aggregation queries on non-managed fields, you must specify a field you defined in the `customFields` argument passed to `UpdateIndexingConfiguration` or **update-indexing-configuration**. If the field value is inconsistent with the configured field data type, this value is ignored when you perform an aggregation query.

If a field cannot be indexed because of a mismatched type, the Fleet Indexing service sends an error log to CloudWatch Logs. The error log contains the field name, the value that could not be converted, and the thing name for the device. The following is an example error log:

```
{
  "timestamp": "2017-02-20 20:31:22.932",
  "logLevel": "ERROR",
  "traceId": "79738924-1025-3a00-a669-7bec69f7f07a",
  "accountId": "000000000000",
  "status": "SucceededWithIssues",
  "eventType": "IndexingCustomFieldFailed",
  "thingName": "thing0",
  "failedCustomFields": [
    {
      "Name": "attributeName1",
      "Value": "apple",
      "ExpectedType": "String"
    },
    {
      "Name": "attributeName2",
      "Value": "2",
      "ExpectedType": "Boolean"
    }
  ]
}
```

If a device has been disconnected for approximately an hour, the connectivity status `timestamp` value might be missing. For persistent sessions, the value might be missing after a client has been disconnected longer than the configured time-to-live (TTL) for the persistent session. The connectivity status data is indexed only for connections where the client ID has a matching thing name. (The client ID is the value used to connect a device to AWS IoT Core.)

Troubleshooting fleet indexing configuration

Can't downgrade fleet indexing configuration

Downgrading fleet indexing configuration is not supported when you want to remove the data sources that are associated with a fleet metric or a dynamic group.

For example, if your indexing configuration has registry data, shadow data, and connectivity data, and a fleet metric exists with the query `thingName:TempSensor* AND shadow.desired.temperature>80`, updating the indexing configuration to include only the registry data will result in an error.

Modifying custom fields used by existing fleet metrics is not supported.

Can't update your indexing configuration due to incompatible fleet metrics or dynamic groups

If you can't update your indexing configuration due to incompatible fleet metrics or dynamic groups, delete the incompatible fleet metrics or dynamic groups before you update the indexing configuration.

Troubleshooting location indexing and geoqueries

To troubleshoot mismatched type errors in location indexing and geoqueries, you can enable CloudWatch logs. For more information about how to monitor AWS IoT using CloudWatch, follow [the step-by-step guide].

When you index location data using geoqueries, the location fields you specify in `geoLocations` must match the location fields you pass to `UpdateIndexingConfiguration`. If there's a mismatch, fleet indexing sends a mismatched type error to CloudWatch. The error log contains the field name, the value that could not be converted, and the thing name for the device.

The following is an example error log:

```
{
```

```
"timestamp": "2023-11-09 01:39:43.466",
    "logLevel": "ERROR",
    "traceId": "79738924-1025-3a00-a669-7bec69f7f07a",
    "accountId": "123456789012",
    "status": "Failure",
    "eventType": "IndexingGeoLocationFieldFailed",
    "thingName": "thing0",
    "failedGeolocationFields": [
        {
"Name": "attributeName1",
            "Value": "apple",
            "ExpectedType": "Geopoint"
        }
    ],
    "reason": "failed to index the field because it could not be converted to one of
 the expected geoLocation formats."
}
```


For more information, see Indexing location data.

Troubleshooting fleet metrics

Can't see data points in CloudWatch

If you're able to create a fleet metric but you can't see data points in CloudWatch, it's likely that you don't have a thing that meets the query string criteria.

See this example command of how to create a fleet metric:

```
aws iot create-fleet-metric --metric-name "example_FM" --query-string
 "thingName:TempSensor* AND attributes.temperature>80" --period 60 --aggregation-field
 "attributes.temperature" --aggregation-type name=Statistics,values=count
```

If you don't have a thing that meets the query string criteria `--query-string "thingName:TempSensor* AND attributes.temperature>80"`:

- With `values=count`, you'll be able to create a fleet metric and there'll be data points to show in CloudWatch. The data points of the value `count` is always 0.
- With `values` other than `count`, you'll be able to create a fleet metric but you won't see the fleet metric in CloudWatch and there'll be no data points to show in CloudWatch.

AWS IoT Device Management Software Package Catalog Troubleshooting

This is the troubleshooting section for AWS IoT Device Management Software Package Catalog.

General Troubleshooting Error Messages

This section lists common errors seen throughout the software package version lifecycle.

HeadBucket errors

The following error messages appear when calling the HeadBucket API operation or head-bucket CLI command to validate the Amazon S3 bucket used for file upload during a job deployment.

For more information on using an Amazon S3 bucket for uploading files during a job deployment, see Presigned URL for file upload.

```
InvalidRoleException
    "Permission denied when attempting to use role %s to access bucket %s."
```

```
InvalidRequestException
    "Cross region S3 bucket is not supported for presigned url upload placeholder"
```

```
InvalidRequestException
    "S3 bucket in job document presigned url upload placeholder not found"
```

```
InvalidRequestException
    "Given S3 bucket name is invalid."
```

```
InvalidRequestException
    "Provided S3 bucket is not valid: %s. Error: %s"
```

Amazon S3 GetObject

The following error message occurs when an invalid argument is provided, thus causing the Amazon S3 `GetObject` API operation to fail.

```
InvalidRequestException
    "Provided argument for presigned url is invalid"
```

Amazon S3 Version ID Support

When requesting access to an Amazon S3 bucket using versioning control, make sure to include your `versionId` or the below error may populate.

For more information on Amazon S3 buckets using versioning control, see Using versioning in Amazon S3 buckets

```
InvalidRequestException
    "VersionId not found when attempting to access s3 url"
```

Placeholders inside of a Presigned URL for file upload

The following error messages appear when encountering issues with a placeholder inside of a presigned URL used for uploading files to a destination Amazon S3 bucket during a job deployment. For more information on using an Amazon S3 bucket for uploading files during a job deployment and what a local placeholder is, see Presigned URL for file upload.

The below error message appears when the local placeholder is not recognized.

```
InvalidJobDocumentException
    "Undefined placeholder, ${...}, inside of presign url upload parameter"
```

The below error message appears when attempting to use the local placeholder in a presigned URL not meant for file upload.

```
InvalidJobDocumentException
    "Local placeholder, ${...}, is only valid inside of presign url upload"
```

Amazon S3 URL Nested Incorrectly

The following error message appears when the Amazon S3 URL is incorrectly nested inside of another placeholder.

```
InvalidJobDocumentException
```

```
    "${aws:%s[...]} should not be the second layer pattern."
```

Package Version Artifact Nesting

The following error message appears when the package version artifact presigned URL is incorrectly nested inside of another placeholder.

```
InvalidJobDocumentException
    "${aws:iot:package:[...]:artifact:s3-presigned-url} cannot be nested inside another
 placeholder."
```

Missing Package Version Artifact

The following error message appears when the referenced package version artifact is not found.

```
InvalidJobDocumentException
    "Package %s version %s does not have an associated artifact to generate an S3
 presigned url."
```

Software Package and Package Verion Placeholders

The following error message appears when the job document placeholder for software package and package version can't resolve to the desired valid values for the job deployment due to multiple software packages and package versions referenced in the `destinationPackageVersions` parameter or the *Version ARN* tab on the *Package Version* details page.

```
InvalidJobDocumentException
    "Cannot resolve empty package name and version name given multiple elements in
 destination package versions."
```

Using Empty Software Package and Package Version

The following error message appears when you attempt to attempt to use an empty package or package version without the other in a job document.

```
InvalidJobDocumentException
    "Empty package name and version name have to be used in pair."
```

Null Use in Job Document

The following error message appears when you attempt to specify `$null` as a package version in the job document. `$null` can only be used inside of the `destinationPackageVersions` parameter when using the `CreateJob` API operation.

```
InvalidJobDocumentException
    "$null is not allowed to be referenced as a package version in job documents."
```

All Attributes in a Package Version

The following error message appears when you attempt to use all attributes in a package version and surround it with additional text or placeholders.

For more information on using all attributes in a software package version, see Substitution parameters for AWS IoT jobs

```
InvalidJobDocumentException
    "The package version attribute placeholder for all attributes has to be a
 json value by itself and not appended with other strings or nested with other
 placeholders."
```

Local Placeholder Limit in Presigned URL for File Upload

The following error message appears when you exceed the limit for number of local placeholders used in a presigned URL for file upload during a job deployment.

For more information on using a presigned URL for file upload during a job deployment, see Presigned URL for file upload

```
InvalidJobDocumentException
    "The occurrence of local placeholder %s within S3 presigned url upload placeholder
 exceeds limit of %d."
```

Local Placeholders in an Amazon S3 Bucket

The following error message appears when you attempt to place a local placeholder URL in the Amazon S3 bucket name for a presigned URL placeholder used for file upload during a job deployment.

For more information on using a presigned URL for file upload during a job deployment, see [Presigned URL for file upload]

```
InvalidJobDocumentException
    "S3 bucket name in presigned url upload is not allowed to contain any placeholders"
```

Opening and Closing Brackets

The following error message appears when you add a parameter or placeholder to a job document without a closing brace "}".

```
InvalidJobDocumentException
    "One or more parameters or placeholders are not terminated."
```

IAM role with Amazon S3 Presigned URL

The following error message appears when you attempt to use an Amazon S3 presigned URL in a job document without an IAM role.

For more information on Amazon S3 presigned URLs, see [Working with presigned URLs].

```
InvalidRequestException
    "presignedUrlConfig role ARN is required to generate an S3 presigned url in job
 document."
```

IAM role with Amazon S3 Presigned URL for Package Version Artifact

The following error message appears when you attempt to use an Amazon S3 presigned URL representing a package version artifact in a job document without an IAM role.

```
InvalidRequestException
    "presignedUrlConfig role ARN is required to generate an S3 presigned url in job
 document for package %s version %s artifact."
```

Software Bill of Materials Error Messages

This section lists common errors associated with a software bill of materials (SBOM) linked to a package version.

Input Validation for SBOM Association Request

The following error message appears when using the `AssociateSbomWithPackageVersion` API operation and the `s3Location` parameter is null.

```
InvalidRequestException "Associate request needs to include SBOM reference"
```

For more information on the `AssociateSbomWithPackageVersion` API operation, see AssociateSbomWithPackageVersion.

SBOM Validation Errors

This section lists common errors seen during the initial validaiton of the software bill of materials (SBOM) when associated with a software package version.

The following error message appears when using the `AssociateSbomWithPackageVersion` API operation and `bucket` in the `s3Location` parameter is null.

```
InvalidRequestException "S3 bucket name for SBOM cannot be null"
```

The following error message appears when the string in `bucket` in the `s3Location` parameter for the `AssociateSbomWithPackageVersion` API operation is too long.

```
InvalidRequestException "S3 bucket name for SBOM is illegal. String length exceeds
 limit"
```

The following error message appears when the `key` parameter is null.

```
InvalidRequestException "S3 key name for SBOM cannot be null"
```

The following error message appears when the string in `key` in the `s3Location` parameter for the `AssociateSbomWithPackageVersion` API operation is too long.

```
InvalidRequestException "S3 key name for SBOM is illegal. String length exceeds limit"
```

The following error message appears when the string in `version` in the `s3Location` parameter for the `AssociateSbomWithPackageVersion` API operation is null.

```
InvalidRequestException "S3 object version for SBOM cannot be null"
```

The following error message appears when the string in `version` in the `s3Location` parameter for the `AssociateSbomWithPackageVersion` API operation is too long.

```
InvalidRequestException "S3 object version for SBOM is illegal. String length exceeds
 limit"
```

The following error message appears when the the size of the SBOM zip archive file stored in the Amazon S3 bucket is too big.

```
InvalidRequestException "S3 object file size exceeds limit"
```

The following error message appears when you use the `AssociateSbomWithPackageVersion` API operation and the current number of SBOM validations in progress is already at the maximum limit.

```
LimitExceededException "Too many ongoing SBOM validation workflows. Please wait and
 retry"
```

Access Issues with SBOM File in Amazon S3 bucket

The following error message appears when another entity fails to access the Amazon S3 bucket due to the Amazon S3 bucket not existing or the proper permissions have not been granted for accessing the Amazon S3 bucket.

For more information on the required permissions policy for accessing the Amazon S3 bucket, see Software Bill of Materials Storage.

```
InvalidRequestException "SBOM not accessible by the service. Please make sure the
 bucket exists and S3 permission is granted."
```

The following error message appears when another entity fails to access the SBOM zip archive file in the `key` parameter due to the Amazon S3 bucket not existing or the proper permissions have not been granted for accessing content stored in the Amazon S3 bucket.

```
InvalidRequestException "SBOM not accessible by the service. Please make sure the key
 exists and S3 permission is granted."
```

The following error message appears when another entity fails to access the Amazon S3 bucket due to the bucket, key, and version ID not existing or the proper permissions have not been granted for accessing the Amazon S3 bucket. Additionally, this error message can appear if the permissions granted are insufficient for accessing the SBOM zip archive file in the Amazon S3 bucket.

```
InvalidRequestException "SBOM not accessible by the service. Please make sure the
 bucket/key/version exists and S3 permission is granted."
```

The following error message appears when another entity fails to access the Amazon S3 bucket due to the bucket being located in another region.

```
InvalidRequestException "Cross-region S3 bucket for %s is not supported."
```

The following error message appears when another entity fails to access the Amazon S3 bucket due to the `bucket`, `key`, or `version` parameters being spelled incorrectly when using the `AssociateSbomWithPackageVersion` API operation.

```
InvalidRequestException "Please make sure SBOM S3 bucket name/key length/version is
 valid"
```

AWS IoT Device Advisor troubleshooting guide

Help us improve this topic

Let us know what would help make it better

General

Q: Can I run multiple test suites in parallel?

> A: Yes. Device Advisor now supports running multiple test suites on different devices using a Device-level endpoint. If you use the Account-level endpoint, you can run one suite at a time because one Device Advisor endpoint is available per account. For more information see Configure your device.

Q: I saw from my device that the TLS connection was denied by Device Advisor. Is this expected?

> A: Yes. Device Advisor denies the TLS connection before and after each test run. We recommend that users implement a device retry mechanism to have a fully automated testing experience

with Device Advisor. If you execute a test suite with more than one test case, for example TLS connect, MQTT connect, and MQTT publish, then we recommend that you have a mechanism built for your device. The mechanism can try to connect to our test endpoint every 5 seconds for a minute to two. In this way you can run multiple test cases in sequence in an automated manner.

Q: Can I get a history of Device Advisor API calls made on my account for security analysis and operational troubleshooting purposes?

A: Yes. To receive a history of Device Advisor API calls made on your account, you simply turn on CloudTrail in the AWS IoT Management Console and filter the event source to be `iotdeviceadvisor.amazonaws.com`.

Q: How do I view Device Advisor logs in CloudWatch?

A: Logs generated during a test suite run are uploaded to CloudWatch if you add the required policy (for example, **CloudWatchFullAccess**) to your service role (see Setting up). If there is at least one test case in the test suite, a log group "aws/iot/deviceadvisor/$testSuiteId" is created with two log streams. One stream is the "$testRunId" and includes logs of actions taken before and after executing the test cases in your test suite, such as setup and cleanup steps. The other log stream is "$suiteRunId_$testRunId," which is specific to a test suite run. Events sent from devices and AWS IoT Core will be logged to this log stream.

Q: What is the purpose of the device permission role?

A: Device Advisor stands between your test device and AWS IoT Core to simulate test scenarios. It accepts connections and messages from your test devices and forwards them to AWS IoT Core by assuming your device permission role and initiating a connection on your behalf. It's important to make sure the device role permissions are the same as those on the certificate you use for running tests. AWS IoT certificate policies are not enforced when Device Advisor initiates a connection to AWS IoT Core on your behalf by using the device permission role. However, the permissions from the device permission role you set are enforced.

Q: In what Regions is Device Advisor supported?

A: Device Advisor is supported in us-east-1, us-west-2, ap-northeast-1, and eu-west-1 Regions.

Q: Why do I see inconsistent results?

A: One of the primary causes of inconsistent results is setting a test's `EXECUTION_TIMEOUT` to a value that is too low. For more information about recommended and default `EXECUTION_TIMEOUT` values, see Device Advisor test cases.

Q: What MQTT protocol does Device Advisor support?

A: Device Advisor supports MQTT Version 3.1.1 with X509 client certificates.

Q: What if my test case failed with an execution timed out message even though I tried to connect my device to the test endpoint?

A: Validate all the steps under Create an IAM role to be used as your device role. If the test still fails, it could be that the device is not sending the correct Server Name Indication (SNI) extension, which is required for Device Advisor to work. The correct SNI value is the endpoint address returned when following the Configure your device section. AWS IoT also requires devices to send the Server Name Indication (SNI) extension to the Transport Layer Security (TLS) protocol. For more information, see Transport security in AWS IoT.

Q: My MQTT connection fails with an "libaws-c-mqtt: AWS_ERROR_MQTT_UNEXPECTED_HANGUP" error (or) my device's MQTT connection is being automatically disconnected from the Device Advisor endpoint. How can this error be resolved?

A: This particular error code and unexpected disconnections can be caused by many different things, but is most likely related to the device role attached to the device. The below checkpoints (in order of priority) will resolve this issue.

- The device role attached to the device must have the minimum IAM permissions required to run the tests. Device Advisor will use the attached device role to perform AWS IoT MQTT actions on behalf of the test device. If required permissions are absent, then the `AWS_ERROR_MQTT_UNEXPECTED_HANGUP` error will be seen or unexpected disconnections will happen while the device tries to connect to Device Advisor endpoint. For example, if you selected to run the **MQTT Publish** test case, both Connect and Publish actions must be included in the role with the corresponding ClientId and Topic (you can provide multiple values by using commas to separate the values, and you can provide prefix values using a wildcard (*) character. For example: To provide permissions to publish on any topic beginning with `TestTopic`, you can provide "`TestTopic*`" as the resource value. Here are some examples of policies.
- Mismatch between the values defined in the device role for your resource types and the actual values used in code. For example: A mismatch in ClientId defined in the role and the actual ClientId used in your device code. Values like ClientId, Topic, and TopicFilter must be identical in the device role and code.
- The device certificate attached to your device must be active and have a policy attached to it with the required action permissions for resources. Note that, the device certificate policy grants or denies access to AWS IoT resources and AWS IoT Core data plane operations. Device

Advisor requires you to have an active device certificate attached to your device which grants the action permissions used during a test case.

AWS IoT errors

Help us improve this topic

Let us know what would help make it better

This section lists the error codes sent by AWS IoT.

Message broker error codes

Error code	Error description
400	Bad request.
401	Unauthorized.
403	Forbidden.
426	Upgrade required.
503	Service unavailable.

Identity and security error codes

Error code	Error description
401	Unauthorized.

Device shadow error codes

Error code	Error description
400	Bad request.
401	Unauthorized.

Error code	Error description
403	Forbidden.
404	Not found.
409	Conflict.
413	Request too large.
422	Failed to process request.
429	Too many requests.
500	Internal error.
503	Service unavailable.

AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client

This page summarizes the AWS IoT Device SDKs, open-source libraries, developer guides, sample apps, and porting guides to help you build innovative IoT solutions with AWS IoT and your choice of hardware platforms.

These SDKs are for use on your IoT device. If you're developing an IoT app for use on a mobile device, see the AWS Mobile SDKs. If you're developing an IoT app or server-side program, see the AWS SDKs.

AWS IoT Device SDKs

The AWS IoT Device SDKs include open-source libraries, developer guides with samples, and porting guides so that you can build innovative IoT products or solutions on your choice of hardware platforms.

> **Note**
>
> The AWS IoT Device SDKs have released an MQTT 5 client. The AWS IoT Device SDKs don't support using TLS 1.3 on macOS.

These SDKs help you connect your IoT devices to AWS IoT using the MQTT and WSS protocols.

C++

AWS IoT C++ Device SDK

The AWS IoT C++ Device SDK allows developers to build connected applications using AWS and the AWS IoT APIs. Specifically, this SDK was designed for devices that are not resource constrained and require advanced features such as message queuing, multi-threading support, and the latest language features. For more information, see the following:

- AWS IoT Device SDK C++ v2 on GitHub
- AWS IoT Device SDK C++ v2 Readme
- AWS IoT Device SDK C++ v2 Samples

- AWS IoT Device SDK C++ v2 API documentation

Python

AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python makes it possible for developers to write Python scripts to use their devices to access the AWS IoT platform through MQTT or MQTT over the WebSocket protocol. By connecting their devices to AWS IoT, users can securely work with the message broker, rules, and shadows provided by AWS IoT and with other AWS services like AWS Lambda, Kinesis, and Amazon S3, and more.

- AWS IoT Device SDK for Python v2 on GitHub
- AWS IoT Device SDK for Python v2 Readme
- AWS IoT Device SDK for Python v2 Samples
- AWS IoT Device SDK for Python v2 API documentation

JavaScript

AWS IoT Device SDK for JavaScript

The aws-iot-device-sdk.js package makes it possible for developers to write JavaScript applications that access AWS IoT using MQTT or MQTT over the WebSocket protocol. It can be used in Node.js environments and browser applications. For more information, see the following:

- AWS IoT Device SDK for JavaScript v2 on GitHub
- AWS IoT Device SDK for JavaScript v2 Readme
- AWS IoT Device SDK for JavaScript v2 Samples
- AWS IoT Device SDK for JavaScript v2 API documentation

Java

AWS IoT Device SDK for Java

The AWS IoT Device SDK for Java makes it possible for Java developers to access the AWS IoT platform through MQTT or MQTT over the WebSocket protocol. The SDK is built with shadow support. You can access shadows by using HTTP methods, including GET, UPDATE, and DELETE. The SDK also supports a simplified shadow access model, which allows developers to exchange data with shadows by just using getter and setter methods, without having to serialize or deserialize any JSON documents.

> **Note**
>
> The AWS IoT Device SDK for Java v2 now supports Android development. For more information, see AWS IoT Device SDK for Android.

For more information, see the following:

- AWS IoT Device SDK for Java v2 on GitHub
- AWS IoT Device SDK for Java v2 Readme
- AWS IoT Device SDK for Java v2 Samples
- AWS IoT Device SDK for Java v2 API documentation

Swift

AWS IoT Device SDK for Swift

The AWS IoT Device SDK for Swift makes it possible for Swift developers to create AWS IoT applications for Linux and Apple macOS, iOS, and tvOS platforms using the MQTT 5 protocol.

For more information, see the following:

- AWS IoT Device SDK for Swift on GitHub
- AWS IoT Device SDK for Swift Readme
- AWS IoT Device SDK for Swift Samples

AWS IoT Device SDK for Embedded C

> **Note**
>
> This SDK is intended for use by experienced embedded-software developers.

The AWS IoT Device SDK for Embedded C (C-SDK) is a collection of C source files under the MIT open source license that can be used in embedded applications to securely connect IoT devices to AWS IoT Core. It includes an MQTT client, JSON Parser, and AWS IoT Device Shadow, AWS IoT Jobs, AWS IoT Fleet Provisioning, and AWS IoT Device Defender libraries. This SDK is distributed in source form and can be built into customer firmware along with application code, other libraries, and an operating system (OS) of your choice.

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs).

For more information, see the following:

- AWS IoT Device SDK for Embedded C on GitHub
- AWS IoT Device SDK for Embedded C Readme
- AWS IoT Device SDK for Embedded C Samples

AWS Mobile SDKs

The AWS Mobile SDKs provide mobile app developers platform-specific support for the APIs of the AWS IoT Core services, IoT device communication using MQTT, and the APIs of other AWS services.

Android

AWS Mobile SDK for Android

The AWS Mobile SDK for Android contains a library, samples, and documentation for developers to build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- AWS Mobile SDK for Android on GitHub
- AWS Mobile SDK for Android Readme
- AWS Mobile SDK for Android Samples
- AWS Mobile SDK for Android API reference
- AWSIoTClient Class reference documentation

iOS

AWS Mobile SDK for iOS

The AWS Mobile SDK for iOS is an open-source software development kit, distributed under an Apache Open Source license. The AWS Mobile SDK for iOS provides a library, code samples, and documentation to help developers build connected mobile applications using AWS. This SDK also includes support for MQTT device communications and calling the APIs of the AWS IoT Core services. For more information, see the following:

- AWS Mobile SDK for iOS on GitHub
- AWS Mobile SDK for iOS Readme
- AWS Mobile SDK for iOS Samples
- AWSIoT Class reference docs in the AWS Mobile SDK for iOS

AWS IoT Device Client

The AWS IoT Device Client provides code to help your device connect to AWS IoT, perform fleet provisioning tasks, support device security policies, connect using secure tunneling, and process jobs on your device. You can install this software on your device to handle these routine device tasks so you can focus on your specific solution.

> **Note**
>
> The AWS IoT Device Client works with microprocessor-based IoT devices with x86_64 or ARM processors and common Linux operating systems.

C++

AWS IoT Device Client

For more information about the AWS IoT Device Client in C++, see the following:

- AWS IoT Device Client in C++ source code on GitHub
- AWS IoT Device Client in C++ Readme

Earlier AWS IoT Device SDKs versions

These are earlier versions of AWS IoT Device SDKs that have been replaced by the newer versions listed above. These SDKs are receiving only maintenance and security updates. They will not be updated to include new features and should not be used on new projects.

- AWS IoT C++ Device SDK on GitHub
- AWS IoT C++ Device SDK Readme
- AWS IoT Device SDK for Python v1 on GitHub
- AWS IoT Device SDK for Python v1 Readme
- AWS IoT Device SDK for Java on GitHub
- AWS IoT Device SDK for Java Readme
- AWS IoT Device SDK for JavaScript on GitHub
- AWS IoT Device SDK for JavaScript Readme
- Arduino Yún SDK on GitHub
- Arduino Yún SDK Readme

Code examples for AWS IoT using AWS SDKs

The following code examples show how to use AWS IoT with an AWS software development kit (SDK).

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

Basic examples for AWS IoT using AWS SDKs

The following code examples show how to use the basics of AWS IoT with AWS SDKs.

Examples

Hello AWS IoT

The following code examples show how to get started using AWS IoT.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/dotnetv4/IoT#code-examples).

```
/// <summary>
/// Hello AWS IoT example.
/// </summary>
public class HelloIoT
{
    /// <summary>
    /// Main method to run the Hello IoT example.
    /// </summary>
    /// <param name="args">Command line arguments.</param>
    /// <returns>A Task object.</returns>
    public static async Task Main(string[] args)
    {
        var iotClient = new AmazonIoTClient();

        try
        {
            Console.WriteLine("Hello AWS IoT! Let's list your IoT Things:");
            Console.WriteLine(new string('-', 80));

            // Use pages of 10.
            var request = new ListThingsRequest()
            {
                MaxResults = 10
            };
            var response = await iotClient.ListThingsAsync(request);

            // Since there is not a built-in paginator, use the NextMarker to
 paginate.
            bool hasMoreResults = true;

            var things = new List<ThingAttribute>();
            while (hasMoreResults)
```

```
            {
                things.AddRange(response.Things);

                // If NextMarker is not null, there are more results. Get the
next page of results.
                if (!String.IsNullOrEmpty(response.NextMarker))
                {
                    request.Marker = response.NextMarker;
                    response = await iotClient.ListThingsAsync(request);
                }
                else
                    hasMoreResults = false;
            }

            if (things is { Count: > 0 })
            {
                Console.WriteLine($"Found {things.Count} IoT Things:");
                foreach (var thing in things)
                {
                    Console.WriteLine($"- Thing Name: {thing.ThingName}");
                    Console.WriteLine($"  Thing ARN: {thing.ThingArn}");
                    Console.WriteLine($"  Thing Type: {thing.ThingTypeName ?? "No
type specified"}");
                    Console.WriteLine($"  Version: {thing.Version}");

                    if (thing.Attributes?.Count > 0)
                    {
                        Console.WriteLine("  Attributes:");
                        foreach (var attr in thing.Attributes)
                        {
                            Console.WriteLine($"    {attr.Key}: {attr.Value}");
                        }
                    }
                    Console.WriteLine();
                }
            }
            else
            {
                Console.WriteLine("No IoT Things found in your account.");
                Console.WriteLine("You can create IoT Things using the IoT Basics
scenario example.");
            }

            Console.WriteLine("Hello IoT completed successfully.");
```

```
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            Console.WriteLine($"Request throttled, please try again later:
 {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Couldn't list Things. Here's why: {ex.Message}");
        }
    }
}
```

- For API details, see listThings in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Code for the CMakeLists.txt CMake file.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS iot)

# Set this project's name.
project("hello_iot")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
```

```
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
    # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

    # set(BIN_SUB_DIR "/Debug") # If you are building from the command line, you
 may need to uncomment this
    # and set the proper subdirectory to the executables' location.

    AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_iot.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code for the hello_iot.cpp source file.

```
#include <aws/core/Aws.h>
#include <aws/iot/IoTClient.h>
#include <aws/iot/model/ListThingsRequest.h>
#include <iostream>

/*
 *  A "Hello IoT" starter application which initializes an AWS IoT client and
 *  lists the AWS IoT topics in the current account.
 *
 *  main function
 *
 *  Usage: 'hello_iot'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
```

```
    //  Optional: change the log level for debugging.
    //  options.loggingOptions.logLevel = Aws::Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::IoT::IoTClient iotClient(clientConfig);
        // List the things in the current account.
        Aws::IoT::Model::ListThingsRequest listThingsRequest;

        Aws::String nextToken; // Used for pagination.
        Aws::Vector<Aws::IoT::Model::ThingAttribute> allThings;

        do {
            if (!nextToken.empty()) {
                listThingsRequest.SetNextToken(nextToken);
            }

            Aws::IoT::Model::ListThingsOutcome listThingsOutcome =
 iotClient.ListThings(
                    listThingsRequest);
            if (listThingsOutcome.IsSuccess()) {
                const Aws::Vector<Aws::IoT::Model::ThingAttribute> &things =
 listThingsOutcome.GetResult().GetThings();
                allThings.insert(allThings.end(), things.begin(), things.end());
                nextToken = listThingsOutcome.GetResult().GetNextToken();
            }
            else {
                std::cerr << "List things failed"
                          << listThingsOutcome.GetError().GetMessage() <<
 std::endl;
                break;
            }
        } while (!nextToken.empty());

        std::cout << allThings.size() << " thing(s) found." << std::endl;
        for (auto const &thing: allThings) {
            std::cout << thing.GetThingName() << std::endl;
        }
    }

    Aws::ShutdownAPI(options); // Should only be called once.
```

```
    return 0;
}
```

- For API details, see listThings in *AWS SDK for C++ API Reference*.

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iot.IotClient;
import software.amazon.awssdk.services.iot.model.ListThingsRequest;
import software.amazon.awssdk.services.iot.model.ListThingsResponse;
import software.amazon.awssdk.services.iot.model.ThingAttribute;
import software.amazon.awssdk.services.iot.paginators.ListThingsIterable;

import java.util.List;

public class HelloIoT {
    public static void main(String[] args) {
        System.out.println("Hello AWS IoT. Here is a listing of your AWS IoT
 Things:");
        IotClient iotClient = IotClient.builder()
            .region(Region.US_EAST_1)
            .build();

        listAllThings(iotClient);
    }
```

```
    public static void listAllThings(IotClient iotClient) {
        iotClient.listThingsPaginator(ListThingsRequest.builder()
                .maxResults(10)
                .build())
            .stream()
            .flatMap(response -> response.things().stream())
            .forEach(attribute -> {
                System.out.println("Thing name: " + attribute.thingName());
                System.out.println("Thing ARN: " + attribute.thingArn());
            });
    }
}
```

- For API details, see listThings in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import aws.sdk.kotlin.services.iot.IotClient
import aws.sdk.kotlin.services.iot.model.ListThingsRequest

suspend fun main() {
    println("A listing of your AWS IoT Things:")
    listAllThings()
}

suspend fun listAllThings() {
    val thingsRequest =
        ListThingsRequest {
            maxResults = 10
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
```

```
        val response = iotClient.listThings(thingsRequest)
        val thingList = response.things
        if (thingList != null) {
            for (attribute in thingList) {
                println("Thing name ${attribute.thingName}")
                println("Thing ARN: ${attribute.thingArn}")
            }
        }
    }
}
```

- For API details, see listThings in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
def hello_iot():
    """
    Use the AWS SDK for Python (Boto3) to create an AWS IoT client and list
    up to 10 things in your AWS IoT account.
    This example uses the default settings specified in your shared credentials
    and config files.
    """
    try:
        iot_client = boto3.client("iot")
        response = iot_client.list_things(maxResults=10)
        things = response.get("things", [])

        print("Hello, AWS IoT! Here are your things:")
        if things:
            for i, thing in enumerate(things, 1):
                print(f"{i}. {thing['thingName']}")
        else:
            print("No things found in your AWS IoT account.")
```

```
    except ClientError as e:
        if e.response["Error"]["Code"] == "UnauthorizedException":
            print("You don't have permission to access AWS IoT.")
        else:
            print(f"Couldn't access AWS IoT. Error: {e}")
    except NoCredentialsError:
        print("No AWS credentials found. Please configure your credentials.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

- For API details, see listThings in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Learn the basics of AWS IoT with an AWS SDK

The following code examples show how to:

- Create an AWS IoT Thing.
- Generate a device certificate.
- Update an AWS IoT Thing with Attributes.
- Return a unique endpoint.
- List your AWS IoT certificates.
- Update an AWS IoT shadow.
- Write out state information.
- Creates a rule.
- List your rules.
- Search things using the Thing name.
- Delete an AWS IoT Thing.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

Run an interactive scenario demonstrating AWS IoT features.

```
/// <summary>
/// Scenario class for AWS IoT basics.
/// </summary>
public class IoTBasics
{
    public static bool IsInteractive = true;
    public static IoTWrapper? Wrapper = null;
    public static IAmazonCloudFormation? CloudFormationClient = null;
    public static ILogger<IoTBasics> logger = null!;
    private static IoTWrapper _iotWrapper = null!;
    private static IAmazonCloudFormation _amazonCloudFormation = null!;
    private static ILogger<IoTBasics> _logger = null!;

    private static string _stackName = "IoTBasicsStack";
    private static string _stackResourcePath = "../../../../../../scenarios/
basics/iot/iot_usecase/resources/cfn_template.yaml";

    /// <summary>
    /// Main method for the IoT Basics scenario.
    /// </summary>
    /// <param name="args">Command line arguments.</param>
    /// <returns>A Task object.</returns>
    public static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonIoT>(new AWSOptions() { Region =
 RegionEndpoint.USEast1 })
                    .AddAWSService<IAmazonCloudFormation>()
                        .AddTransient<IoTWrapper>()
```

```
                        .AddLogging(builder => builder.AddConsole())
                        .AddSingleton<IAmazonIotData>(sp =>
                        {
                            var iotService = sp.GetRequiredService<IAmazonIoT>();
                            var request = new DescribeEndpointRequest
                            {
                                EndpointType = "iot:Data-ATS"
                            };
                            var response =
 iotService.DescribeEndpointAsync(request).Result;
                            return new AmazonIotDataClient($"https://
{response.EndpointAddress}/");
                        })
            )
            .Build();

        logger = LoggerFactory.Create(builder => builder.AddConsole())
            .CreateLogger<IoTBasics>();

        Wrapper = host.Services.GetRequiredService<IoTWrapper>();
        CloudFormationClient =
 host.Services.GetRequiredService<IAmazonCloudFormation>();

        // Set the private fields for backwards compatibility
        _logger = logger;
        _iotWrapper = Wrapper;
        _amazonCloudFormation = CloudFormationClient;

        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Welcome to the AWS IoT example scenario.");
        Console.WriteLine("This example program demonstrates various interactions
 with the AWS Internet of Things (IoT) Core service.");
        Console.WriteLine();
        if (IsInteractive)
        {
            Console.WriteLine("Press Enter to continue...");
            Console.ReadLine();
        }
        Console.WriteLine(new string('-', 80));

        try
        {
            await RunScenarioAsync();
        }
```

```
        catch (Exception ex)
        {
            _logger.LogError(ex, "There was a problem running the scenario.");
            Console.WriteLine($"\nAn error occurred: {ex.Message}");
        }

        Console.WriteLine(new string('-', 80));
        Console.WriteLine("The AWS IoT scenario has successfully completed.");
        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Run the IoT Basics scenario.
    /// </summary>
    /// <returns>A Task object.</returns>
    public static async Task RunScenarioAsync()
    {
        // Use static properties if available, otherwise use private fields
        var iotWrapper = Wrapper ?? _iotWrapper;
        var cloudFormationClient = CloudFormationClient ?? _amazonCloudFormation;
        var scenarioLogger = logger ?? _logger;

        await RunScenarioInternalAsync(iotWrapper, cloudFormationClient,
 scenarioLogger);
    }

    /// <summary>
    /// Internal method to run the IoT Basics scenario with injected
 dependencies.
    /// </summary>
    /// <param name="iotWrapper">The IoT wrapper instance.</param>
    /// <param name="cloudFormationClient">The CloudFormation client instance.</
param>
    /// <param name="scenarioLogger">The logger instance.</param>
    /// <returns>A Task object.</returns>
    private static async Task RunScenarioInternalAsync(IoTWrapper iotWrapper,
 IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics> scenarioLogger)
    {
        string thingName = $"iot-thing-{Guid.NewGuid():N}";
        string certificateArn = "";
        string certificateId = "";
        string ruleName = $"iotruledefault";
        string snsTopicArn = "";
```

```
        try
        {
            // Step 1: Create an AWS IoT Thing
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("1. Create an AWS IoT Thing.");
            Console.WriteLine("An AWS IoT Thing represents a virtual entity in 
 the AWS IoT service that can be associated with a physical device.");
            Console.WriteLine();

            if (IsInteractive)
            {
                Console.Write("Enter Thing name: ");
                var userInput = Console.ReadLine();
                if (!string.IsNullOrEmpty(userInput))
                    thingName = userInput;
            }
            else
            {
                Console.WriteLine($"Using default Thing name: {thingName}");
            }

            var thingArn = await iotWrapper.CreateThingAsync(thingName);
            Console.WriteLine($"{thingName} was successfully created. The ARN 
 value is {thingArn}");
            Console.WriteLine(new string('-', 80));

            // Step 1.1: List AWS IoT Things
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("2. List AWS IoT Things.");
            Console.WriteLine("Now let's list the IoT Things to see the Thing we 
 just created.");
            Console.WriteLine();
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var things = await iotWrapper.ListThingsAsync();
            Console.WriteLine($"Found {things.Count} IoT Things:");
            foreach (var thing in things.Take(10)) // Show first 10 things
            {
                Console.WriteLine($"Thing Name: {thing.ThingName}");
                Console.WriteLine($"Thing ARN: {thing.ThingArn}");
```

```
                if (thing.Attributes != null && thing.Attributes.Any())
                {
                    Console.WriteLine("Attributes:");
                    foreach (var attr in thing.Attributes)
                    {
                        Console.WriteLine($"  {attr.Key}: {attr.Value}");
                    }
                }
                Console.WriteLine("--------------");
            }
            Console.WriteLine();
            Console.WriteLine(new string('-', 80));

            // Step 2: Generate a Device Certificate
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("3. Generate a device certificate.");
            Console.WriteLine("A device certificate performs a role in securing 
 the communication between devices (Things) and the AWS IoT platform.");
            Console.WriteLine();

            var createCert = "y";
            if (IsInteractive)
            {
                Console.Write($"Do you want to create a certificate for 
 {thingName}? (y/n)");
                createCert = Console.ReadLine();
            }
            else
            {
                Console.WriteLine($"Creating certificate for {thingName}...");
            }

            if (createCert?.ToLower() == "y")
            {
                var certificateResult = await 
 iotWrapper.CreateKeysAndCertificateAsync();
                if (certificateResult.HasValue)
                {
                    var (certArn, certPem, certId) = certificateResult.Value;
                    certificateArn = certArn;
                    certificateId = certId;

                    Console.WriteLine($"\nCertificate:");
                    // Show only first few lines of certificate for brevity
```

```
                    var lines = certPem.Split('\n');
                    for (int i = 0; i < Math.Min(lines.Length, 5); i++)
                    {
                        Console.WriteLine(lines[i]);
                    }
                    if (lines.Length > 5)
                    {
                        Console.WriteLine("...");
                    }

                    Console.WriteLine($"\nCertificate ARN:");
                    Console.WriteLine(certificateArn);

                    // Step 3: Attach the Certificate to the AWS IoT Thing
                    Console.WriteLine("Attach the certificate to the AWS IoT
 Thing.");
                    var attachResult = await
 iotWrapper.AttachThingPrincipalAsync(thingName, certificateArn);
                    if (attachResult)
                    {
                        Console.WriteLine("Certificate attached to Thing
 successfully.");
                    }
                    else
                    {
                        Console.WriteLine("Failed to attach certificate to
 Thing.");
                    }

                    Console.WriteLine("Thing Details:");
                    Console.WriteLine($"Thing Name: {thingName}");
                    Console.WriteLine($"Thing ARN: {thingArn}");
                }
                else
                {
                    Console.WriteLine("Failed to create certificate.");
                }
            }
            Console.WriteLine(new string('-', 80));

            // Step 4: Update an AWS IoT Thing with Attributes
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("4. Update an AWS IoT Thing with Attributes.");
```

```
            Console.WriteLine("IoT Thing attributes, represented as key-value
 pairs, offer a pivotal advantage in facilitating efficient data");
            Console.WriteLine("management and retrieval within the AWS IoT
 ecosystem.");
            Console.WriteLine();
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var attributes = new Dictionary<string, string>
            {
                { "Location", "Seattle" },
                { "DeviceType", "Sensor" },
                { "Firmware", "1.2.3" }
            };

            await iotWrapper.UpdateThingAsync(thingName, attributes);
            Console.WriteLine("Thing attributes updated successfully.");
            Console.WriteLine(new string('-', 80));

            // Step 5: Return a unique endpoint specific to the Amazon Web
 Services account
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("5. Return a unique endpoint specific to the Amazon
 Web Services account.");
            Console.WriteLine();
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var endpoint = await iotWrapper.DescribeEndpointAsync();
            if (endpoint != null)
            {
                var subdomain = endpoint.Split('.')[0];
                Console.WriteLine($"Extracted subdomain: {subdomain}");
                Console.WriteLine($"Full Endpoint URL: https://{endpoint}");
            }
            else
            {
                Console.WriteLine("Failed to retrieve endpoint.");
```

```
            }
            Console.WriteLine(new string('-', 80));

            // Step 6: List your AWS IoT certificates
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("6. List your AWS IoT certificates");
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var certificates = await iotWrapper.ListCertificatesAsync();
            foreach (var cert in certificates.Take(5)) // Show first 5
 certificates
            {
                Console.WriteLine($"Cert id: {cert.CertificateId}");
                Console.WriteLine($"Cert Arn: {cert.CertificateArn}");
            }
            Console.WriteLine();
            Console.WriteLine(new string('-', 80));

            // Step 7: Create an IoT shadow
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("7. Update an IoT shadow that refers to a digital
 representation or virtual twin of a physical IoT device");
            Console.WriteLine();
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var shadowPayload = JsonSerializer.Serialize(new
            {
                state = new
                {
                    desired = new
                    {
                        temperature = 25,
                        humidity = 50
                    }
                }
            });
```

```
            await iotWrapper.UpdateThingShadowAsync(thingName, shadowPayload);
            Console.WriteLine("Thing Shadow updated successfully.");
            Console.WriteLine(new string('-', 80));

            // Step 8: Write out the state information, in JSON format
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("8. Write out the state information, in JSON
 format.");
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var shadowData = await iotWrapper.GetThingShadowAsync(thingName);
            Console.WriteLine($"Received Shadow Data: {shadowData}");
            Console.WriteLine(new string('-', 80));

            // Step 9: Set up resources (SNS topic and IAM role) and create a
 rule
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("9. Set up resources and create a rule");
            Console.WriteLine();

            // Deploy CloudFormation stack to create SNS topic and IAM role
            Console.WriteLine("Deploying CloudFormation stack to create SNS topic
 and IAM role...");

            var deployStack = !IsInteractive || GetYesNoResponse("Would you like
 to deploy the CloudFormation stack? (y/n) ");
            if (deployStack)
            {
                if (IsInteractive)
                {
                    Console.Write(
                        $"Enter stack resource file path (or press Enter for
 default '{_stackResourcePath}'): ");
                    var userResourcePath = Console.ReadLine();
                    if (!string.IsNullOrEmpty(userResourcePath))
                        _stackResourcePath = userResourcePath;
                }

                _stackName = PromptUserForStackName();
```

```
                var deploySuccess = await DeployCloudFormationStack(_stackName,
cloudFormationClient, scenarioLogger);

                if (deploySuccess)
                {
                    // Get stack outputs
                    var stackOutputs = await GetStackOutputs(_stackName,
cloudFormationClient, scenarioLogger);
                    if (stackOutputs != null)
                    {
                        snsTopicArn = stackOutputs["SNSTopicArn"];
                        string roleArn = stackOutputs["RoleArn"];

                        Console.WriteLine($"Successfully deployed stack. SNS
topic: {snsTopicArn}");
                        Console.WriteLine($"Successfully deployed stack. IAM
role: {roleArn}");

                        if (IsInteractive)
                        {
                            Console.Write($"Enter Rule name (press Enter for
default '{ruleName}'): ");
                            var userRuleName = Console.ReadLine();
                            if (!string.IsNullOrEmpty(userRuleName))
                                ruleName = userRuleName;
                        }
                        else
                        {
                            Console.WriteLine($"Using default rule name:
{ruleName}");
                        }

                        // Now create the IoT rule with the CloudFormation
outputs
                        var ruleResult = await
iotWrapper.CreateTopicRuleAsync(ruleName, snsTopicArn, roleArn);
                        if (ruleResult)
                        {
                            Console.WriteLine("IoT Rule created successfully.");
                        }
                        else
                        {
                            Console.WriteLine("Failed to create IoT rule.");
```

```
                        }
                    }
                    else
                    {
                        Console.WriteLine("Failed to get stack outputs. Skipping
 rule creation.");
                    }
                }
                else
                {
                    Console.WriteLine("Failed to deploy CloudFormation stack.
 Skipping rule creation.");
                }
            }
            else
            {
                Console.WriteLine("Skipping CloudFormation stack deployment and
 rule creation.");
            }
            Console.WriteLine(new string('-', 80));

            // Step 10: List your rules
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("10. List your rules.");
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var rules = await iotWrapper.ListTopicRulesAsync();
            Console.WriteLine("List of IoT Rules:");
            foreach (var rule in rules.Take(5)) // Show first 5 rules
            {
                Console.WriteLine($"Rule Name: {rule.RuleName}");
                Console.WriteLine($"Rule ARN: {rule.RuleArn}");
                Console.WriteLine("--------------");
            }
            Console.WriteLine();
            Console.WriteLine(new string('-', 80));

            // Step 11: Search things using the Thing name
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("11. Search things using the Thing name.");
```

```
            if (IsInteractive)
            {
                Console.WriteLine("Press Enter to continue...");
                Console.ReadLine();
            }

            var searchResults = await iotWrapper.SearchIndexAsync($"thingName:
{thingName}");
            if (searchResults.Any())
            {
                Console.WriteLine($"Thing id found using search is
 {searchResults.First().ThingId}");
            }
            else
            {
                Console.WriteLine($"No search results found for Thing:
 {thingName}");
            }
            Console.WriteLine(new string('-', 80));

            // Step 12: Cleanup - Detach and delete certificate
            if (!string.IsNullOrEmpty(certificateArn))
            {
                Console.WriteLine(new string('-', 80));
                var deleteCert = "y";
                if (IsInteractive)
                {
                    Console.Write($"Do you want to detach and delete the
 certificate for {thingName}? (y/n)");
                    deleteCert = Console.ReadLine();
                }
                else
                {
                    Console.WriteLine($"Detaching and deleting certificate for
 {thingName}...");
                }

                if (deleteCert?.ToLower() == "y")
                {
                    Console.WriteLine("12. You selected to detach and delete the
 certificate.");
                    if (IsInteractive)
                    {
                        Console.WriteLine("Press Enter to continue...");
```

```
                        Console.ReadLine();
                    }

                    await iotWrapper.DetachThingPrincipalAsync(thingName, 
certificateArn);
                    Console.WriteLine($"{certificateArn} was successfully removed 
from {thingName}");

                    await iotWrapper.DeleteCertificateAsync(certificateId);
                    Console.WriteLine($"{certificateArn} was successfully 
deleted.");
                }
                Console.WriteLine(new string('-', 80));
            }

            // Step 13: Delete the AWS IoT Thing
            Console.WriteLine(new string('-', 80));
            Console.WriteLine("13. Delete the AWS IoT Thing.");
            var deleteThing = "y";
            if (IsInteractive)
            {
                Console.Write($"Do you want to delete the IoT Thing? (y/n)");
                deleteThing = Console.ReadLine();
            }
            else
            {
                Console.WriteLine($"Deleting IoT Thing {thingName}...");
            }

            if (deleteThing?.ToLower() == "y")
            {
                await iotWrapper.DeleteThingAsync(thingName);
                Console.WriteLine($"Deleted Thing {thingName}");
            }
            Console.WriteLine(new string('-', 80));

            // Step 14: Clean up CloudFormation stack
            if (!string.IsNullOrEmpty(snsTopicArn))
            {
                Console.WriteLine(new string('-', 80));
                Console.WriteLine("14. Clean up CloudFormation stack.");
                Console.WriteLine("Deleting the CloudFormation stack and all 
resources...");
```

```
                var cleanup = !IsInteractive || GetYesNoResponse("Do you want to
 delete the CloudFormation stack and all resources? (y/n) ");
                if (cleanup)
                {
                    var ruleCleanupSuccess = await
 iotWrapper.DeleteTopicRuleAsync(ruleName);

                    var stackCleanupSuccess = await
 DeleteCloudFormationStack(_stackName, cloudFormationClient, scenarioLogger);
                    if (ruleCleanupSuccess && stackCleanupSuccess)
                    {
                        Console.WriteLine("Successfully cleaned up CloudFormation
 stack and all resources.");
                    }
                    else
                    {
                        Console.WriteLine("Some cleanup operations failed. Check
 the logs for details.");
                    }
                }
                else
                {
                    Console.WriteLine($"Resources will remain. Stack name:
 {_stackName}");
                }
                Console.WriteLine(new string('-', 80));
            }
        }
        catch (Exception ex)
        {
            scenarioLogger.LogError(ex, "Error occurred during scenario
 execution.");

            // Cleanup on error
            if (!string.IsNullOrEmpty(certificateArn) && !
string.IsNullOrEmpty(thingName))
            {
                try
                {
                    await iotWrapper.DetachThingPrincipalAsync(thingName,
 certificateArn);
                    await iotWrapper.DeleteCertificateAsync(certificateId);
                }
                catch (Exception cleanupEx)
```

```
                {
                    scenarioLogger.LogError(cleanupEx, "Error during cleanup.");
                }
            }

            if (!string.IsNullOrEmpty(thingName))
            {
                try
                {
                    await iotWrapper.DeleteThingAsync(thingName);
                }
                catch (Exception cleanupEx)
                {
                    scenarioLogger.LogError(cleanupEx, "Error during Thing
 cleanup.");
                }
            }

            // Clean up CloudFormation stack on error
            if (!string.IsNullOrEmpty(snsTopicArn))
            {
                try
                {
                    await _iotWrapper.DeleteTopicRuleAsync(ruleName);
                    await DeleteCloudFormationStack(_stackName,
 cloudFormationClient, scenarioLogger);
                }
                catch (Exception cleanupEx)
                {
                    scenarioLogger.LogError(cleanupEx, "Error during
 CloudFormation stack cleanup.");
                }
            }

            throw;
        }
    }

    /// <summary>
    /// Deploys the CloudFormation stack with the necessary resources.
    /// </summary>
    /// <param name="stackName">The name of the CloudFormation stack.</param>
    /// <param name="cloudFormationClient">The CloudFormation client.</param>
    /// <param name="scenarioLogger">The logger.</param>
```

```
    /// <returns>True if the stack was deployed successfully.</returns>
    private static async Task<bool> DeployCloudFormationStack(string stackName,
IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics> scenarioLogger)
    {
        Console.WriteLine($"\nDeploying CloudFormation stack: {stackName}");

        try
        {
            var request = new CreateStackRequest
            {
                StackName = stackName,
                TemplateBody = await File.ReadAllTextAsync(_stackResourcePath),
                Capabilities = new List<string>
{ Capability.CAPABILITY_NAMED_IAM }
            };

            var response = await cloudFormationClient.CreateStackAsync(request);

            if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
            {
                Console.WriteLine($"CloudFormation stack creation started:
{stackName}");

                bool stackCreated = await
WaitForStackCompletion(response.StackId, cloudFormationClient, scenarioLogger);

                if (stackCreated)
                {
                    Console.WriteLine("CloudFormation stack created
successfully.");
                    return true;
                }
                else
                {
                    scenarioLogger.LogError($"CloudFormation stack creation
failed: {stackName}");
                    return false;
                }
            }
            else
            {
                scenarioLogger.LogError($"Failed to create CloudFormation stack:
{stackName}");
                return false;
```

```
            }
        }
        catch (AlreadyExistsException)
        {
            scenarioLogger.LogWarning($"CloudFormation stack '{stackName}'
 already exists. Please provide a unique name.");
            var newStackName = PromptUserForStackName();
            return await DeployCloudFormationStack(newStackName,
 cloudFormationClient, scenarioLogger);
        }
        catch (Exception ex)
        {
            scenarioLogger.LogError(ex, $"An error occurred while deploying the
 CloudFormation stack: {stackName}");
            return false;
        }
    }

    /// <summary>
    /// Waits for the CloudFormation stack to be in the CREATE_COMPLETE state.
    /// </summary>
    /// <param name="stackId">The ID of the CloudFormation stack.</param>
    /// <param name="cloudFormationClient">The CloudFormation client.</param>
    /// <param name="scenarioLogger">The logger.</param>
    /// <returns>True if the stack was created successfully.</returns>
    private static async Task<bool> WaitForStackCompletion(string stackId,
 IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics> scenarioLogger)
    {
        int retryCount = 0;
        const int maxRetries = 30;
        const int retryDelay = 10000;

        while (retryCount < maxRetries)
        {
            var describeStacksRequest = new DescribeStacksRequest
            {
                StackName = stackId
            };

            var describeStacksResponse = await
 cloudFormationClient.DescribeStacksAsync(describeStacksRequest);

            if (describeStacksResponse.Stacks.Count > 0)
            {
```

```
                if (describeStacksResponse.Stacks[0].StackStatus ==
StackStatus.CREATE_COMPLETE)
                {
                    return true;
                }
                if (describeStacksResponse.Stacks[0].StackStatus ==
StackStatus.CREATE_FAILED ||
                    describeStacksResponse.Stacks[0].StackStatus ==
StackStatus.ROLLBACK_COMPLETE)
                {
                    return false;
                }
            }

            Console.WriteLine("Waiting for CloudFormation stack creation to
complete...");
            await Task.Delay(retryDelay);
            retryCount++;
        }

        scenarioLogger.LogError("Timed out waiting for CloudFormation stack
creation to complete.");
        return false;
    }

    /// <summary>
    /// Gets the outputs from the CloudFormation stack.
    /// </summary>
    /// <param name="stackName">The name of the CloudFormation stack.</param>
    /// <param name="cloudFormationClient">The CloudFormation client.</param>
    /// <param name="scenarioLogger">The logger.</param>
    /// <returns>A dictionary of stack outputs.</returns>
    private static async Task<Dictionary<string, string>?> GetStackOutputs(string
stackName, IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics>
scenarioLogger)
    {
        try
        {
            var describeStacksRequest = new DescribeStacksRequest
            {
                StackName = stackName
            };
```

```
            var response = await 
cloudFormationClient.DescribeStacksAsync(describeStacksRequest);

            if (response.Stacks.Count > 0)
            {
                var outputs = new Dictionary<string, string>();
                foreach (var output in response.Stacks[0].Outputs)
                {
                    outputs[output.OutputKey] = output.OutputValue;
                }
                return outputs;
            }

            return null;
        }
        catch (Exception ex)
        {
            scenarioLogger.LogError(ex, $"Failed to get stack outputs for 
{stackName}");
            return null;
        }
    }

    /// <summary>
    /// Deletes the CloudFormation stack and waits for confirmation.
    /// </summary>
    /// <param name="stackName">The name of the CloudFormation stack.</param>
    /// <param name="cloudFormationClient">The CloudFormation client.</param>
    /// <param name="scenarioLogger">The logger.</param>
    /// <returns>True if the stack was deleted successfully.</returns>
    private static async Task<bool> DeleteCloudFormationStack(string stackName, 
IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics> scenarioLogger)
    {
        try
        {
            var request = new DeleteStackRequest
            {
                StackName = stackName
            };

            await cloudFormationClient.DeleteStackAsync(request);
            Console.WriteLine($"CloudFormation stack '{stackName}' is being 
deleted. This may take a few minutes.");
```

```
            bool stackDeleted = await WaitForStackDeletion(stackName,
cloudFormationClient, scenarioLogger);

            if (stackDeleted)
            {
                Console.WriteLine($"CloudFormation stack '{stackName}' has been
deleted.");
                return true;
            }
            else
            {
                scenarioLogger.LogError($"Failed to delete CloudFormation stack
'{stackName}'.");
                return false;
            }
        }
        catch (Exception ex)
        {
            scenarioLogger.LogError(ex, $"An error occurred while deleting the
CloudFormation stack: {stackName}");
            return false;
        }
    }

    /// <summary>
    /// Waits for the stack to be deleted.
    /// </summary>
    /// <param name="stackName">The name of the CloudFormation stack.</param>
    /// <param name="cloudFormationClient">The CloudFormation client.</param>
    /// <param name="scenarioLogger">The logger.</param>
    /// <returns>True if the stack was deleted successfully.</returns>
    private static async Task<bool> WaitForStackDeletion(string stackName,
IAmazonCloudFormation cloudFormationClient, ILogger<IoTBasics> scenarioLogger)
    {
        int retryCount = 0;
        const int maxRetries = 30;
        const int retryDelay = 10000;

        while (retryCount < maxRetries)
        {
            var describeStacksRequest = new DescribeStacksRequest
            {
                StackName = stackName
            };
```

```
            try
            {
                var describeStacksResponse = await
cloudFormationClient.DescribeStacksAsync(describeStacksRequest);

                if (describeStacksResponse.Stacks.Count == 0 ||
                    describeStacksResponse.Stacks[0].StackStatus ==
StackStatus.DELETE_COMPLETE)
                {
                    return true;
                }
            }
            catch (AmazonCloudFormationException ex) when (ex.ErrorCode ==
"ValidationError")
            {
                return true;
            }

            Console.WriteLine($"Waiting for CloudFormation stack '{stackName}' to
be deleted...");
            await Task.Delay(retryDelay);
            retryCount++;
        }

        scenarioLogger.LogError($"Timed out waiting for CloudFormation stack
'{stackName}' to be deleted.");
        return false;
    }

    /// <summary>
    /// Helper method to get a yes or no response from the user.
    /// </summary>
    private static bool GetYesNoResponse(string question)
    {
        Console.WriteLine(question);
        var ynResponse = Console.ReadLine();
        var response = ynResponse != null && ynResponse.Equals("y",
StringComparison.InvariantCultureIgnoreCase);
        return response;
    }

    /// <summary>
    /// Prompts the user for a stack name.
```

```
    /// </summary>
    private static string PromptUserForStackName()
    {
        if (IsInteractive)
        {
            Console.Write($"Enter a name for the CloudFormation stack (press
 Enter for default '{_stackName}'): ");
            string? input = Console.ReadLine();
            if (!string.IsNullOrWhiteSpace(input))
            {
                var regex = new System.Text.RegularExpressions.Regex("[a-zA-Z][-
a-zA-Z0-9]*");
                if (!regex.IsMatch(input))
                {
                    Console.WriteLine($"Invalid stack name. Using default:
 {_stackName}");
                    return _stackName;
                }
                return input;
            }
        }
        return _stackName;
    }
}
```

A wrapper class for AWS IoT SDK methods.

```
/// <summary>
/// Wrapper methods to use Amazon IoT Core with .NET.
/// </summary>
public class IoTWrapper
{
    private readonly IAmazonIoT _amazonIoT;
    private readonly IAmazonIotData _amazonIotData;
    private readonly ILogger<IoTWrapper> _logger;

    /// <summary>
    /// Constructor for the IoT wrapper.
    /// </summary>
    /// <param name="amazonIoT">The injected IoT client.</param>
    /// <param name="amazonIotData">The injected IoT Data client.</param>
    /// <param name="logger">The injected logger.</param>
```

```
    public IoTWrapper(IAmazonIoT amazonIoT, IAmazonIotData amazonIotData,
 ILogger<IoTWrapper> logger)
    {
        _amazonIoT = amazonIoT;
        _amazonIotData = amazonIotData;
        _logger = logger;
    }

    /// <summary>
    /// Creates an AWS IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing to create.</param>
    /// <returns>The ARN of the Thing created, or null if creation failed.</
returns>
    public async Task<string?> CreateThingAsync(string thingName)
    {
        try
        {
            var request = new CreateThingRequest
            {
                ThingName = thingName
            };

            var response = await _amazonIoT.CreateThingAsync(request);
            _logger.LogInformation($"Created Thing {thingName} with ARN
 {response.ThingArn}");
            return response.ThingArn;
        }
        catch (Amazon.IoT.Model.ResourceAlreadyExistsException ex)
        {
            _logger.LogWarning($"Thing {thingName} already exists:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create Thing {thingName}. Here's why:
 {ex.Message}");
            return null;
        }
    }

    /// <summary>
    /// Creates a device certificate for AWS IoT.
```

```
    /// </summary>
    /// <returns>The certificate details including ARN and certificate PEM, or
 null if creation failed.</returns>
    public async Task<(string CertificateArn, string CertificatePem, string
 CertificateId)?> CreateKeysAndCertificateAsync()
    {
        try
        {
            var request = new CreateKeysAndCertificateRequest
            {
                SetAsActive = true
            };

            var response = await
 _amazonIoT.CreateKeysAndCertificateAsync(request);
            _logger.LogInformation($"Created certificate with ARN
 {response.CertificateArn}");
            return (response.CertificateArn, response.CertificatePem,
 response.CertificateId);
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create certificate. Here's why:
 {ex.Message}");
            return null;
        }
    }

    /// <summary>
    /// Attaches a certificate to an IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
    /// <param name="certificateArn">The ARN of the certificate to attach.</
param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> AttachThingPrincipalAsync(string thingName, string
 certificateArn)
    {
```

```
        try
        {
            var request = new AttachThingPrincipalRequest
            {
                ThingName = thingName,
                Principal = certificateArn
            };

            await _amazonIoT.AttachThingPrincipalAsync(request);
            _logger.LogInformation($"Attached certificate {certificateArn} to
 Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot attach certificate - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't attach certificate to Thing. Here's why:
 {ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Updates an IoT Thing with attributes.
    /// </summary>
    /// <param name="thingName">The name of the Thing to update.</param>
    /// <param name="attributes">Dictionary of attributes to add.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> UpdateThingAsync(string thingName, Dictionary<string,
 string> attributes)
    {
        try
        {
            var request = new UpdateThingRequest
            {
                ThingName = thingName,
                AttributePayload = new AttributePayload
                {
                    Attributes = attributes,
```

```
                    Merge = true
                }
            };

            await _amazonIoT.UpdateThingAsync(request);
            _logger.LogInformation($"Updated Thing {thingName} with attributes");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot update Thing - resource not found: 
{ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't update Thing attributes. Here's why: 
{ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Gets the AWS IoT endpoint URL.
    /// </summary>
    /// <returns>The endpoint URL, or null if retrieval failed.</returns>
    public async Task<string?> DescribeEndpointAsync()
    {
        try
        {
            var request = new DescribeEndpointRequest
            {
                EndpointType = "iot:Data-ATS"
            };

            var response = await _amazonIoT.DescribeEndpointAsync(request);
            _logger.LogInformation($"Retrieved endpoint: 
{response.EndpointAddress}");
            return response.EndpointAddress;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later: 
{ex.Message}");
```

```
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't describe endpoint. Here's why:
{ex.Message}");
            return null;
        }
    }

    /// <summary>
    /// Lists all certificates associated with the account.
    /// </summary>
    /// <returns>List of certificate information, or empty list if listing
failed.</returns>
    public async Task<List<Certificate>> ListCertificatesAsync()
    {
        try
        {
            var request = new ListCertificatesRequest();
            var response = await _amazonIoT.ListCertificatesAsync(request);

            _logger.LogInformation($"Retrieved {response.Certificates.Count}
certificates");
            return response.Certificates;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
{ex.Message}");
            return new List<Certificate>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't list certificates. Here's why:
{ex.Message}");
            return new List<Certificate>();
        }
    }

    /// <summary>
    /// Updates the Thing's shadow with new state information.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
```

```
    /// <param name="shadowPayload">The shadow payload in JSON format.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> UpdateThingShadowAsync(string thingName, string
 shadowPayload)
    {
        try
        {
            var request = new UpdateThingShadowRequest
            {
                ThingName = thingName,
                Payload = new
 MemoryStream(System.Text.Encoding.UTF8.GetBytes(shadowPayload))
            };

            await _amazonIotData.UpdateThingShadowAsync(request);
            _logger.LogInformation($"Updated shadow for Thing {thingName}");
            return true;
        }
        catch (Amazon.IotData.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot update Thing shadow - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't update Thing shadow. Here's why:
 {ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Gets the Thing's shadow information.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
    /// <returns>The shadow data as a string, or null if retrieval failed.</
returns>
    public async Task<string?> GetThingShadowAsync(string thingName)
    {
        try
        {
            var request = new GetThingShadowRequest
            {
```

```
                ThingName = thingName
            };

            var response = await _amazonIotData.GetThingShadowAsync(request);
            using var reader = new StreamReader(response.Payload);
            var shadowData = await reader.ReadToEndAsync();

            _logger.LogInformation($"Retrieved shadow for Thing {thingName}");
            return shadowData;
        }
        catch (Amazon.IotData.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot get Thing shadow - resource not found:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't get Thing shadow. Here's why:
 {ex.Message}");
            return null;
        }
    }

    /// <summary>
    /// Creates an IoT topic rule.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="snsTopicArn">The ARN of the SNS topic for the action.</
param>
    /// <param name="roleArn">The ARN of the IAM role.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> CreateTopicRuleAsync(string ruleName, string
 snsTopicArn, string roleArn)
    {
        try
        {
            var request = new CreateTopicRuleRequest
            {
                RuleName = ruleName,
                TopicRulePayload = new TopicRulePayload
                {
                    Sql = "SELECT * FROM 'topic/subtopic'",
                    Description = $"Rule created by .NET example: {ruleName}",
```

```
                    Actions = new List<Amazon.IoT.Model.Action>
                    {
                        new Amazon.IoT.Model.Action
                        {
                            Sns = new SnsAction
                            {
                                TargetArn = snsTopicArn,
                                RoleArn = roleArn
                            }
                        }
                    },
                    RuleDisabled = false
                }
            };

            await _amazonIoT.CreateTopicRuleAsync(request);
            _logger.LogInformation($"Created IoT rule {ruleName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceAlreadyExistsException ex)
        {
            _logger.LogWarning($"Rule {ruleName} already exists: {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create topic rule. Here's why:
 {ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Deletes an IoT topic rule.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DeleteTopicRuleAsync(string ruleName)
    {
        try
        {
            var request = new DeleteTopicRuleRequest
            {
                RuleName = ruleName,
```

```
            };

            await _amazonIoT.DeleteTopicRuleAsync(request);
            _logger.LogInformation($"Deleted IoT rule {ruleName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogWarning($"Rule {ruleName} not found: {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't delete topic rule. Here's why:
{ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Lists all IoT topic rules.
    /// </summary>
    /// <returns>List of topic rules, or empty list if listing failed.</returns>
    public async Task<List<TopicRuleListItem>> ListTopicRulesAsync()
    {
        try
        {
            var request = new ListTopicRulesRequest();
            var response = await _amazonIoT.ListTopicRulesAsync(request);

            _logger.LogInformation($"Retrieved {response.Rules.Count} IoT
rules");
            return response.Rules;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
{ex.Message}");
            return new List<TopicRuleListItem>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't list topic rules. Here's why:
{ex.Message}");
```

```
            return new List<TopicRuleListItem>();
        }
    }

    /// <summary>
    /// Searches for IoT Things using the search index.
    /// </summary>
    /// <param name="queryString">The search query string.</param>
    /// <returns>List of Things that match the search criteria, or empty list if
search failed.</returns>
    public async Task<List<ThingDocument>> SearchIndexAsync(string queryString)
    {
        try
        {
            // First, try to perform the search
            var request = new SearchIndexRequest
            {
                QueryString = queryString
            };

            var response = await _amazonIoT.SearchIndexAsync(request);
            _logger.LogInformation($"Search found {response.Things.Count}
Things");
            return response.Things;
        }
        catch (Amazon.IoT.Model.IndexNotReadyException ex)
        {
            _logger.LogWarning($"Search index not ready, setting up indexing
configuration: {ex.Message}");
            return await SetupIndexAndRetrySearchAsync(queryString);
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex) when
(ex.Message.Contains("index") || ex.Message.Contains("Index"))
        {
            _logger.LogWarning($"Search index not configured, setting up indexing
configuration: {ex.Message}");
            return await SetupIndexAndRetrySearchAsync(queryString);
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
{ex.Message}");
            return new List<ThingDocument>();
        }
```

```
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't search index. Here's why: {ex.Message}");
            return new List<ThingDocument>();
        }
    }

    /// <summary>
    /// Sets up the indexing configuration and retries the search after waiting
 for the index to be ready.
    /// </summary>
    /// <param name="queryString">The search query string.</param>
    /// <returns>List of Things that match the search criteria, or empty list if
 setup/search failed.</returns>
    private async Task<List<ThingDocument>> SetupIndexAndRetrySearchAsync(string
 queryString)
    {
        try
        {
            // Update indexing configuration to REGISTRY mode
            _logger.LogInformation("Setting up IoT search indexing
 configuration...");
            await _amazonIoT.UpdateIndexingConfigurationAsync(
                new UpdateIndexingConfigurationRequest()
                {
                    ThingIndexingConfiguration = new ThingIndexingConfiguration()
                    {
                        ThingIndexingMode = ThingIndexingMode.REGISTRY
                    }
                });

            _logger.LogInformation("Indexing configuration updated. Waiting for
 index to be ready...");

            // Wait for the index to be set up - this can take some time
            const int maxRetries = 10;
            const int retryDelaySeconds = 10;

            for (int attempt = 1; attempt <= maxRetries; attempt++)
            {
                try
                {
                    _logger.LogInformation($"Waiting for index to be ready
 (attempt {attempt}/{maxRetries})...");
```

```
                    await Task.Delay(TimeSpan.FromSeconds(retryDelaySeconds));

                    // Try to get the current indexing configuration to see if
it's ready
                    var configResponse = await
_amazonIoT.GetIndexingConfigurationAsync(new GetIndexingConfigurationRequest());
                    if
(configResponse.ThingIndexingConfiguration?.ThingIndexingMode ==
ThingIndexingMode.REGISTRY)
                    {
                        // Try the search again
                        var request = new SearchIndexRequest
                        {
                            QueryString = queryString
                        };

                        var response = await
_amazonIoT.SearchIndexAsync(request);
                        _logger.LogInformation($"Search found
{response.Things.Count} Things after index setup");
                        return response.Things;
                    }
                }
                catch (Amazon.IoT.Model.IndexNotReadyException)
                {
                    // Index still not ready, continue waiting
                    _logger.LogInformation("Index still not ready, continuing to
wait...");
                    continue;
                }
                catch (Amazon.IoT.Model.InvalidRequestException ex) when
(ex.Message.Contains("index") || ex.Message.Contains("Index"))
                {
                    // Index still not ready, continue waiting
                    _logger.LogInformation("Index still not ready, continuing to
wait...");
                    continue;
                }
            }

            _logger.LogWarning("Timeout waiting for search index to be ready
after configuration update");
            return new List<ThingDocument>();
        }
```

```
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't set up search index configuration. Here's
 why: {ex.Message}");
            return new List<ThingDocument>();
        }
    }

    /// <summary>
    /// Detaches a certificate from an IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
    /// <param name="certificateArn">The ARN of the certificate to detach.</
param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DetachThingPrincipalAsync(string thingName, string
 certificateArn)
    {
        try
        {
            var request = new DetachThingPrincipalRequest
            {
                ThingName = thingName,
                Principal = certificateArn
            };

            await _amazonIoT.DetachThingPrincipalAsync(request);
            _logger.LogInformation($"Detached certificate {certificateArn} from
 Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot detach certificate - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't detach certificate from Thing. Here's
 why: {ex.Message}");
            return false;
        }
    }
```

```
    /// <summary>
    /// Deletes an IoT certificate.
    /// </summary>
    /// <param name="certificateId">The ID of the certificate to delete.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DeleteCertificateAsync(string certificateId)
    {
        try
        {
            // First, update the certificate to inactive state
            var updateRequest = new UpdateCertificateRequest
            {
                CertificateId = certificateId,
                NewStatus = CertificateStatus.INACTIVE
            };
            await _amazonIoT.UpdateCertificateAsync(updateRequest);

            // Then delete the certificate
            var deleteRequest = new DeleteCertificateRequest
            {
                CertificateId = certificateId
            };

            await _amazonIoT.DeleteCertificateAsync(deleteRequest);
            _logger.LogInformation($"Deleted certificate {certificateId}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot delete certificate - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't delete certificate. Here's why:
 {ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Deletes an IoT Thing.
```

```
    /// </summary>
    /// <param name="thingName">The name of the Thing to delete.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DeleteThingAsync(string thingName)
    {
        try
        {
            var request = new DeleteThingRequest
            {
                ThingName = thingName
            };

            await _amazonIoT.DeleteThingAsync(request);
            _logger.LogInformation($"Deleted Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot delete Thing - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't delete Thing. Here's why: {ex.Message}");
            return false;
        }
    }

    /// <summary>
    /// Lists IoT Things with pagination support.
    /// </summary>
    /// <returns>List of Things, or empty list if listing failed.</returns>
    public async Task<List<ThingAttribute>> ListThingsAsync()
    {
        try
        {
            // Use pages of 10.
            var request = new ListThingsRequest()
            {
                MaxResults = 10
            };
            var response = await _amazonIoT.ListThingsAsync(request);
```

```
            // Since there is not a built-in paginator, use the NextMarker to
 paginate.
            bool hasMoreResults = true;

            var things = new List<ThingAttribute>();
            while (hasMoreResults)
            {
                things.AddRange(response.Things);

                // If NextMarker is not null, there are more results. Get the
 next page of results.
                if (!String.IsNullOrEmpty(response.NextMarker))
                {
                    request.Marker = response.NextMarker;
                    response = await _amazonIoT.ListThingsAsync(request);
                }
                else
                    hasMoreResults = false;
            }

            _logger.LogInformation($"Retrieved {things.Count} Things");
            return things;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
 {ex.Message}");
            return new List<ThingAttribute>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't list Things. Here's why: {ex.Message}");
            return new List<ThingAttribute>();
        }
    }

}
```

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/cpp/example_code/iot#code-examples).

Create an AWS IoT thing.

```
    Aws::String thingName = askQuestion("Enter a thing name: ");

    if (!createThing(thingName, clientConfiguration)) {
        std::cerr << "Exiting because createThing failed." << std::endl;
        cleanup("", "", "", "", "", false, clientConfiguration);
        return false;
    }
```

```
//! Create an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::createThing(const Aws::String &thingName,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::CreateThingRequest createThingRequest;
    createThingRequest.SetThingName(thingName);

    Aws::IoT::Model::CreateThingOutcome outcome = iotClient.CreateThing(
            createThingRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to create thing " << thingName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
```

```
    }

    return outcome.IsSuccess();
}
```

Generate and attach a device certificate.

```
    Aws::String certificateARN;
    Aws::String certificateID;
    if (askYesNoQuestion("Would you like to create a certificate for your thing?
 (y/n) ")) {
        Aws::String outputFolder;
        if (askYesNoQuestion(
                "Would you like to save the certificate and keys to file? (y/n)
 ")) {
            outputFolder = std::filesystem::current_path();
            outputFolder += "/device_keys_and_certificates";

            std::filesystem::create_directories(outputFolder);

            std::cout << "The certificate and keys will be saved to the folder: "
                      << outputFolder << std::endl;
        }

        if (!createKeysAndCertificate(outputFolder, certificateARN,
 certificateID,
                                      clientConfiguration)) {
            std::cerr << "Exiting because createKeysAndCertificate failed."
                      << std::endl;
            cleanup(thingName, "", "", "", "", false, clientConfiguration);
            return false;
        }

        std::cout << "\nNext, the certificate will be attached to the thing.\n"
                  << std::endl;
        if (!attachThingPrincipal(certificateARN, thingName,
 clientConfiguration)) {
            std::cerr << "Exiting because attachThingPrincipal failed." <<
 std::endl;
            cleanup(thingName, certificateARN, certificateID, "", "",
                    false,
                    clientConfiguration);
```

```
            return false;
        }
    }
```

```
//! Create keys and certificate for an Aws IoT device.
//! This routine will save certificates and keys to an output folder, if
 provided.
/*!
  \param outputFolder: Location for storing output in files, ignored when string
 is empty.
  \param certificateARNResult: A string to receive the ARN of the created
 certificate.
  \param certificateID: A string to receive the ID of the created certificate.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::createKeysAndCertificate(const Aws::String &outputFolder,
                                           Aws::String &certificateARNResult,
                                           Aws::String &certificateID,
                                           const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient client(clientConfiguration);
    Aws::IoT::Model::CreateKeysAndCertificateRequest
 createKeysAndCertificateRequest;

    Aws::IoT::Model::CreateKeysAndCertificateOutcome outcome =
            client.CreateKeysAndCertificate(createKeysAndCertificateRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created a certificate and keys" << std::endl;
        certificateARNResult = outcome.GetResult().GetCertificateArn();
        certificateID = outcome.GetResult().GetCertificateId();
        std::cout << "Certificate ARN: " << certificateARNResult << ",
 certificate ID: "
                  << certificateID << std::endl;

        if (!outputFolder.empty()) {
            std::cout << "Writing certificate and keys to the folder '" <<
 outputFolder
                      << "'." << std::endl;
            std::cout << "Be sure these files are stored securely." << std::endl;
```

```
            Aws::String certificateFilePath = outputFolder + "/
certificate.pem.crt";
            std::ofstream certificateFile(certificateFilePath);
            if (!certificateFile.is_open()) {
                std::cerr << "Error opening certificate file, '" <<
 certificateFilePath
                          << "'."
                          << std::endl;
                return false;
            }
            certificateFile << outcome.GetResult().GetCertificatePem();
            certificateFile.close();

            const Aws::IoT::Model::KeyPair &keyPair =
 outcome.GetResult().GetKeyPair();

            Aws::String privateKeyFilePath = outputFolder + "/private.pem.key";
            std::ofstream privateKeyFile(privateKeyFilePath);
            if (!privateKeyFile.is_open()) {
                std::cerr << "Error opening private key file, '" <<
 privateKeyFilePath
                          << "'."
                          << std::endl;
                return false;
            }
            privateKeyFile << keyPair.GetPrivateKey();
            privateKeyFile.close();

            Aws::String publicKeyFilePath = outputFolder + "/public.pem.key";
            std::ofstream publicKeyFile(publicKeyFilePath);
            if (!publicKeyFile.is_open()) {
                std::cerr << "Error opening public key file, '" <<
 publicKeyFilePath
                          << "'."
                          << std::endl;
                return false;
            }
            publicKeyFile << keyPair.GetPublicKey();
        }
    }
    else {
        std::cerr << "Error creating keys and certificate: "
                  << outcome.GetError().GetMessage() << std::endl;
    }
```

```
    return outcome.IsSuccess();
}

//! Attach a principal to an AWS IoT thing.
/*!
  \param principal: A principal to attach.
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::attachThingPrincipal(const Aws::String &principal,
                                       const Aws::String &thingName,
                                       const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient client(clientConfiguration);
    Aws::IoT::Model::AttachThingPrincipalRequest request;
    request.SetPrincipal(principal);
    request.SetThingName(thingName);
    Aws::IoT::Model::AttachThingPrincipalOutcome outcome =
 client.AttachThingPrincipal(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully attached principal to thing." << std::endl;
    }
    else {
        std::cerr << "Failed to attach principal to thing." <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

Perform various operations on the AWS IoT thing.

```
    if (!updateThing(thingName, { {"location", "Office"}, {"firmwareVersion",
 "v2.0"} }, clientConfiguration)) {
        std::cerr << "Exiting because updateThing failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }
```

```
    printAsterisksLine();

    std::cout << "Now an endpoint will be retrieved for your account.\n" <<
std::endl;
    std::cout << "An IoT Endpoint refers to a specific URL or Uniform Resource
Locator that serves as the entry point\n"
    << "for communication between IoT devices and the AWS IoT service." <<
std::endl;

    askQuestion("Press Enter to continue:", alwaysTrueTest);

    Aws::String endpoint;
    if (!describeEndpoint(endpoint, clientConfiguration)) {
        std::cerr << "Exiting because getEndpoint failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }
    std::cout <<"Your endpoint is " << endpoint << "." << std::endl;
    printAsterisksLine();

    std::cout << "Now the certificates in your account will be listed." <<
std::endl;
    askQuestion("Press Enter to continue:", alwaysTrueTest);

    if (!listCertificates(clientConfiguration)) {
        std::cerr << "Exiting because listCertificates failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }

    printAsterisksLine();

    std::cout << "Now the shadow for the thing will be updated.\n" << std::endl;
    std::cout << "A thing shadow refers to a feature that enables you to create a
virtual representation, or \"shadow,\"\n"
    << "of a physical device or thing. The thing shadow allows you to synchronize
and control the state of a device between\n"
    << "the cloud and the device itself. and the AWS IoT service. For example,
you can write and retrieve JSON data from a thing shadow." << std::endl;
    askQuestion("Press Enter to continue:", alwaysTrueTest);
```

```
    if (!updateThingShadow(thingName, R"({"state":{"reported":
{"temperature":25,"humidity":50}}})", clientConfiguration)) {
        std::cerr << "Exiting because updateThingShadow failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }

    printAsterisksLine();

    std::cout << "Now, the state information for the shadow will be retrieved.\n"
 << std::endl;
    askQuestion("Press Enter to continue:", alwaysTrueTest);

    Aws::String shadowState;
    if (!getThingShadow(thingName, shadowState, clientConfiguration)) {
        std::cerr << "Exiting because getThingShadow failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }
    std::cout << "The retrieved shadow state is: " << shadowState << std::endl;

    printAsterisksLine();

    std::cout << "A rule with now be added to to the thing.\n" << std::endl;
    std::cout << "Any user who has permission to create rules will be able to
 access data processed by the rule." << std::endl;
    std::cout << "In this case, the rule will use an Simple Notification Service
 (SNS) topic and an IAM rule." << std::endl;
    std::cout << "These resources will be created using a CloudFormation
 template." << std::endl;
    std::cout << "Stack creation may take a few minutes." << std::endl;

    askQuestion("Press Enter to continue: ", alwaysTrueTest);
    Aws::Map<Aws::String, Aws::String> outputs
 =createCloudFormationStack(STACK_NAME,clientConfiguration);
    if (outputs.empty()) {
        std::cerr << "Exiting because createCloudFormationStack failed." <<
 std::endl;
        cleanup(thingName, certificateARN, certificateID, "", "", false,
                clientConfiguration);
        return false;
    }
```

```
    // Retrieve the topic ARN and role ARN from the CloudFormation stack outputs.
    auto topicArnIter = outputs.find(SNS_TOPIC_ARN_OUTPUT);
    auto roleArnIter = outputs.find(ROLE_ARN_OUTPUT);
    if ((topicArnIter == outputs.end()) || (roleArnIter == outputs.end())) {
        std::cerr << "Exiting because output '" << SNS_TOPIC_ARN_OUTPUT <<
        "' or '" << ROLE_ARN_OUTPUT << "'not found in the CloudFormation stack."
 << std::endl;
        cleanup(thingName, certificateARN, certificateID, STACK_NAME, "",
                false,
                clientConfiguration);
        return false;
    }

    Aws::String topicArn = topicArnIter->second;
    Aws::String roleArn = roleArnIter->second;
    Aws::String sqlStatement = "SELECT * FROM '";
    sqlStatement += MQTT_MESSAGE_TOPIC_FILTER;
    sqlStatement += "'";

    printAsterisksLine();

    std::cout << "Now a rule will be created.\n" << std::endl;
    std::cout << "Rules are an administrator-level action. Any user who has
 permission\n"
                 << "to create rules will be able to access data processed by the
 rule." << std::endl;
    std::cout << "In this case, the rule will use an SNS topic" << std::endl;
    std::cout << "and the following SQL statement '" << sqlStatement << "'." <<
 std::endl;
    std::cout << "For more information on IoT SQL, see https://
docs.aws.amazon.com/iot/latest/developerguide/iot-sql-reference.html" <<
 std::endl;
    Aws::String ruleName = askQuestion("Enter a rule name: ");
    if (!createTopicRule(ruleName, topicArn, sqlStatement, roleArn,
 clientConfiguration)) {
        std::cerr << "Exiting because createRule failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, STACK_NAME, "",
                false,
                clientConfiguration);
        return false;
    }

    printAsterisksLine();
```

```
    std::cout << "Now your rules will be listed.\n" << std::endl;
    askQuestion("Press Enter to continue: ", alwaysTrueTest);
    if (!listTopicRules(clientConfiguration)) {
        std::cerr << "Exiting because listRules failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, STACK_NAME, ruleName,
                false,
                clientConfiguration);
        return false;
    }

    printAsterisksLine();
    Aws::String queryString = "thingName:" + thingName;
    std::cout << "Now the AWS IoT fleet index will be queried with the query\n'"
    << queryString << "'.\n" << std::endl;
    std::cout << "For query information, see https://docs.aws.amazon.com/iot/
latest/developerguide/query-syntax.html" << std::endl;

    std::cout << "For this query to work, thing indexing must be enabled in your
 account.\n"
    << "This can be done with the awscli command line by calling 'aws iot update-
indexing-configuration'\n"
       << "or it can be done programmatically." << std::endl;
    std::cout << "For more information, see https://docs.aws.amazon.com/iot/
latest/developerguide/managing-index.html" << std::endl;
    if (askYesNoQuestion("Do you want to enable thing indexing in your account?
 (y/n) "))
    {
        Aws::IoT::Model::ThingIndexingConfiguration thingIndexingConfiguration;

 thingIndexingConfiguration.SetThingIndexingMode(Aws::IoT::Model::ThingIndexingMode::REGI

 thingIndexingConfiguration.SetThingConnectivityIndexingMode(Aws::IoT::Model::ThingConnec
        // The ThingGroupIndexingConfiguration object is ignored if not set.
        Aws::IoT::Model::ThingGroupIndexingConfiguration
 thingGroupIndexingConfiguration;
        if (!updateIndexingConfiguration(thingIndexingConfiguration,
 thingGroupIndexingConfiguration, clientConfiguration)) {
            std::cerr << "Exiting because updateIndexingConfiguration failed." <<
 std::endl;
            cleanup(thingName, certificateARN, certificateID, STACK_NAME,
                    ruleName, false,
                    clientConfiguration);
            return false;
```

```
        }
    }

    if (!searchIndex(queryString, clientConfiguration)) {

        std::cerr << "Exiting because searchIndex failed." << std::endl;
        cleanup(thingName, certificateARN, certificateID, STACK_NAME, ruleName,
                false,
                clientConfiguration);
        return false;
    }
```

```
//! Update an AWS IoT thing with attributes.
/*!
  \param thingName: The name for the thing.
  \param attributeMap: A map of key/value attributes/
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::updateThing(const Aws::String &thingName,
                              const std::map<Aws::String, Aws::String>
 &attributeMap,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::UpdateThingRequest request;
    request.SetThingName(thingName);
    Aws::IoT::Model::AttributePayload attributePayload;
    for (const auto &attribute: attributeMap) {
        attributePayload.AddAttributes(attribute.first, attribute.second);
    }
    request.SetAttributePayload(attributePayload);

    Aws::IoT::Model::UpdateThingOutcome outcome = iotClient.UpdateThing(request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully updated thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to update thing " << thingName << ":" <<
                  outcome.GetError().GetMessage() << std::endl;
    }
```

```
    return outcome.IsSuccess();
}

//! Describe the endpoint specific to the AWS account making the call.
/*!
  \param endpointResult: String to receive the endpoint result.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::describeEndpoint(Aws::String &endpointResult,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::String endpoint;
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DescribeEndpointRequest describeEndpointRequest;
    describeEndpointRequest.SetEndpointType(
            "iot:Data-ATS"); // Recommended endpoint type.

    Aws::IoT::Model::DescribeEndpointOutcome outcome =
 iotClient.DescribeEndpoint(
            describeEndpointRequest);

    if (outcome.IsSuccess()) {
        std::cout << "Successfully described endpoint." << std::endl;
        endpointResult = outcome.GetResult().GetEndpointAddress();
    }
    else {
        std::cerr << "Error describing endpoint" <<
 outcome.GetError().GetMessage()
                  << std::endl;
    }

    return outcome.IsSuccess();
}

//! List certificates registered in the AWS account making the call.
/*!
   \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::listCertificates(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::ListCertificatesRequest request;
```

```
    Aws::Vector<Aws::IoT::Model::Certificate> allCertificates;
    Aws::String marker; // Used to paginate results.
    do {
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::IoT::Model::ListCertificatesOutcome outcome =
 iotClient.ListCertificates(
                request);

        if (outcome.IsSuccess()) {
            const Aws::IoT::Model::ListCertificatesResult &result =
 outcome.GetResult();
            marker = result.GetNextMarker();
            allCertificates.insert(allCertificates.end(),
                                   result.GetCertificates().begin(),
                                   result.GetCertificates().end());
        }
        else {
            std::cerr << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
            return false;
        }
    } while (!marker.empty());

    std::cout << allCertificates.size() << " certificate(s) found." << std::endl;

    for (auto &certificate: allCertificates) {
        std::cout << "Certificate ID: " << certificate.GetCertificateId() <<
 std::endl;
        std::cout << "Certificate ARN: " << certificate.GetCertificateArn()
                  << std::endl;
        std::cout << std::endl;
    }

    return true;
}

//! Update the shadow of an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param document: The state information, in JSON format.
```

```
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::updateThingShadow(const Aws::String &thingName,
                                    const Aws::String &document,
                                    const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoTDataPlane::IoTDataPlaneClient
 iotDataPlaneClient(clientConfiguration);
    Aws::IoTDataPlane::Model::UpdateThingShadowRequest updateThingShadowRequest;
    updateThingShadowRequest.SetThingName(thingName);
    std::shared_ptr<std::stringstream> streamBuf =
 std::make_shared<std::stringstream>(
            document);
    updateThingShadowRequest.SetBody(streamBuf);
    Aws::IoTDataPlane::Model::UpdateThingShadowOutcome outcome =
 iotDataPlaneClient.UpdateThingShadow(
            updateThingShadowRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully updated thing shadow." << std::endl;
    }
    else {
        std::cerr << "Error while updating thing shadow."
                  << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}

//! Get the shadow of an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param documentResult: String to receive the state information, in JSON format.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::getThingShadow(const Aws::String &thingName,
                                 Aws::String &documentResult,
                                 const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoTDataPlane::IoTDataPlaneClient iotClient(clientConfiguration);
    Aws::IoTDataPlane::Model::GetThingShadowRequest request;
    request.SetThingName(thingName);
    auto outcome = iotClient.GetThingShadow(request);
```

```
    if (outcome.IsSuccess()) {
        std::stringstream ss;
        ss << outcome.GetResult().GetPayload().rdbuf();
        documentResult = ss.str();
    }
    else {
        std::cerr << "Error getting thing shadow: " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}

//! Create an AWS IoT rule with an SNS topic as the target.
/*!
  \param ruleName: The name for the rule.
  \param snsTopic: The SNS topic ARN for the action.
  \param sql: The SQL statement used to query the topic.
  \param roleARN: The IAM role ARN for the action.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool
AwsDoc::IoT::createTopicRule(const Aws::String &ruleName,
                             const Aws::String &snsTopicARN, const Aws::String
 &sql,
                             const Aws::String &roleARN,
                             const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::CreateTopicRuleRequest request;
    request.SetRuleName(ruleName);

    Aws::IoT::Model::SnsAction snsAction;
    snsAction.SetTargetArn(snsTopicARN);
    snsAction.SetRoleArn(roleARN);

    Aws::IoT::Model::Action action;
    action.SetSns(snsAction);

    Aws::IoT::Model::TopicRulePayload topicRulePayload;
    topicRulePayload.SetSql(sql);
    topicRulePayload.SetActions({action});
```

```
    request.SetTopicRulePayload(topicRulePayload);
    auto outcome = iotClient.CreateTopicRule(request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created topic rule " << ruleName << "." <<
 std::endl;
    }
    else {
        std::cerr << "Error creating topic rule " << ruleName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }
    return outcome.IsSuccess();
}

//! Lists the AWS IoT topic rules.
/*!
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::listTopicRules(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::ListTopicRulesRequest request;

    Aws::Vector<Aws::IoT::Model::TopicRuleListItem> allRules;
    Aws::String nextToken; // Used for pagination.
    do {
        if (!nextToken.empty()) {
            request.SetNextToken(nextToken);
        }

        Aws::IoT::Model::ListTopicRulesOutcome outcome =
 iotClient.ListTopicRules(
                request);

        if (outcome.IsSuccess()) {
            const Aws::IoT::Model::ListTopicRulesResult &result =
 outcome.GetResult();
            allRules.insert(allRules.end(),
                            result.GetRules().cbegin(),
                            result.GetRules().cend());

            nextToken = result.GetNextToken();
        }
```

```
        else {
            std::cerr << "ListTopicRules error: " <<
                      outcome.GetError().GetMessage() << std::endl;
            return false;
        }

    } while (!nextToken.empty());

    std::cout << "ListTopicRules: " << allRules.size() << " rule(s) found."
              << std::endl;
    for (auto &rule: allRules) {
        std::cout << "  Rule name: " << rule.GetRuleName() << ", rule ARN: "
                  << rule.GetRuleArn() << "." << std::endl;
    }

    return true;
}

//! Query the AWS IoT fleet index.
//! For query information, see https://docs.aws.amazon.com/iot/latest/
developerguide/query-syntax.html
/*!
  \param: query: The query string.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::searchIndex(const Aws::String &query,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::SearchIndexRequest request;
    request.SetQueryString(query);

    Aws::Vector<Aws::IoT::Model::ThingDocument> allThingDocuments;
    Aws::String nextToken; // Used for pagination.
    do {
        if (!nextToken.empty()) {
            request.SetNextToken(nextToken);
        }

        Aws::IoT::Model::SearchIndexOutcome outcome =
 iotClient.SearchIndex(request);
```

```
        if (outcome.IsSuccess()) {
            const Aws::IoT::Model::SearchIndexResult &result =
 outcome.GetResult();
            allThingDocuments.insert(allThingDocuments.end(),
                                     result.GetThings().cbegin(),
                                     result.GetThings().cend());
            nextToken = result.GetNextToken();

        }
        else {
            std::cerr << "Error in SearchIndex: " <<
 outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!nextToken.empty());

    std::cout << allThingDocuments.size() << " thing document(s) found." <<
 std::endl;
    for (const auto thingDocument: allThingDocuments) {
        std::cout << "  Thing name: " << thingDocument.GetThingName() << "."
                  << std::endl;
    }
    return true;
}
```

Clean up resources.

```
bool
AwsDoc::IoT::cleanup(const Aws::String &thingName, const Aws::String
 &certificateARN,
                     const Aws::String &certificateID, const Aws::String
 &stackName,
                     const Aws::String &ruleName, bool askForConfirmation,
                     const Aws::Client::ClientConfiguration &clientConfiguration)
 {
    bool result = true;

    if (!ruleName.empty() && (!askForConfirmation ||
                              askYesNoQuestion("Delete the rule '" + ruleName +
                                               "'? (y/n) "))) {
        result &= deleteTopicRule(ruleName, clientConfiguration);
```

```
    }

    Aws::CloudFormation::CloudFormationClient
 cloudFormationClient(clientConfiguration);

    if (!stackName.empty() && (!askForConfirmation ||
                               askYesNoQuestion(
                                       "Delete the CloudFormation stack '" +
 stackName +
                                       "'? (y/n) "))) {
        result &= deleteStack(stackName, clientConfiguration);
    }

    if (!certificateARN.empty() && (!askForConfirmation ||
                                    askYesNoQuestion("Delete the certificate '" +
                                                     certificateARN + "'? (y/n)
 "))) {
        result &= detachThingPrincipal(certificateARN, thingName,
 clientConfiguration);
        result &= deleteCertificate(certificateID, clientConfiguration);
    }

    if (!thingName.empty() && (!askForConfirmation ||
                               askYesNoQuestion("Delete the thing '" + thingName
 +
                                                "'? (y/n) "))) {
        result &= deleteThing(thingName, clientConfiguration);
    }

    return result;
}
```

```
//! Detach a principal from an AWS IoT thing.
/*!
  \param principal: A principal to detach.
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::detachThingPrincipal(const Aws::String &principal,
                                       const Aws::String &thingName,
```

```
                                        const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::DetachThingPrincipalRequest detachThingPrincipalRequest;
    detachThingPrincipalRequest.SetThingName(thingName);
    detachThingPrincipalRequest.SetPrincipal(principal);

    Aws::IoT::Model::DetachThingPrincipalOutcome outcome =
 iotClient.DetachThingPrincipal(
            detachThingPrincipalRequest);

    if (outcome.IsSuccess()) {
        std::cout << "Successfully detached principal " << principal << " from
 thing "
                  << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to detach principal " << principal << " from thing "
                  << thingName << ": "
                  << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}

//! Delete a certificate.
/*!
  \param certificateID: The ID of a certificate.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::deleteCertificate(const Aws::String &certificateID,
                                    const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::DeleteCertificateRequest request;
    request.SetCertificateId(certificateID);

    Aws::IoT::Model::DeleteCertificateOutcome outcome =
 iotClient.DeleteCertificate(
            request);
```

```
    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted certificate " << certificateID <<
 std::endl;
    }
    else {
        std::cerr << "Error deleting certificate " << certificateID << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}

//! Delete an AWS IoT rule.
/*!
  \param ruleName: The name for the rule.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::deleteTopicRule(const Aws::String &ruleName,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DeleteTopicRuleRequest request;
    request.SetRuleName(ruleName);

    Aws::IoT::Model::DeleteTopicRuleOutcome outcome = iotClient.DeleteTopicRule(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted rule " << ruleName << std::endl;
    }
    else {
        std::cerr << "Failed to delete rule " << ruleName <<
                  ": " << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}

//! Delete an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
```

```
bool AwsDoc::IoT::deleteThing(const Aws::String &thingName,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DeleteThingRequest request;
    request.SetThingName(thingName);
    const auto outcome = iotClient.DeleteThing(request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Error deleting thing " << thingName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario demonstrating AWS IoT features.

```
import java.util.Scanner;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
```

```
 * This Java example performs these tasks:
 *
 * 1. Creates an AWS IoT Thing.
 * 2. Generate and attach a device certificate.
 * 3. Update an AWS IoT Thing with Attributes.
 * 4. Get an AWS IoT Endpoint.
 * 5. List your certificates.
 * 6. Updates the shadow for the specified thing..
 * 7. Write out the state information, in JSON format
 * 8. Creates a rule
 * 9. List rules
 * 10. Search things
 * 11. Detach amd delete the certificate.
 * 12. Delete Thing.
 */
public class IotScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <roleARN> <snsAction>

                Where:
                    roleARN - The ARN of an IAM role that has permission to work
 with AWS IOT.
                    snsAction  - An ARN of an SNS topic.
                """;

        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        IotActions iotActions = new IotActions();
        String thingName;
        String ruleName;
        String roleARN = args[0];
        String snsAction = args[1];
        Scanner scanner = new Scanner(System.in);

        System.out.println(DASHES);
```

```
        System.out.println("Welcome to the AWS IoT basics scenario.");
        System.out.println("""
            This example program demonstrates various interactions with the AWS
 Internet of Things (IoT) Core service. The program guides you through a series
 of steps,
            including creating an IoT Thing, generating a device certificate,
 updating the Thing with attributes, and so on.
            It utilizes the AWS SDK for Java V2 and incorporates functionality
 for creating and managing IoT Things, certificates, rules,
            shadows, and performing searches. The program aims to showcase AWS
 IoT capabilities and provides a comprehensive example for
            developers working with AWS IoT in a Java environment.

            Let's get started...

            """);
        System.out.println(DASHES);

        System.out.println("1. Create an AWS IoT Thing.");
        System.out.println("""
            An AWS IoT Thing represents a virtual entity in the AWS IoT service
 that can be associated with
            a physical device.
            """);
        // Prompt the user for input.
        System.out.print("Enter Thing name: ");
        thingName = scanner.nextLine();
        iotActions.createIoTThing(thingName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Generate a device certificate.");
        System.out.println("""
            A device certificate performs a role in securing the communication
 between devices (Things)
            and the AWS IoT platform.
            """);

        System.out.print("Do you want to create a certificate for " +thingName
 +"? (y/n)");
        String certAns = scanner.nextLine();
        String certificateArn="" ;
        if (certAns != null && certAns.trim().equalsIgnoreCase("y")) {
            certificateArn = iotActions.createCertificate();
```

```
            System.out.println("Attach the certificate to the AWS IoT Thing.");
            iotActions.attachCertificateToThing(thingName, certificateArn);
        } else {
            System.out.println("A device certificate was not created.");
        }
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. Update an AWS IoT Thing with Attributes.");
        System.out.println("""
             IoT Thing attributes, represented as key-value pairs, offer a
pivotal advantage in facilitating efficient data
             management and retrieval within the AWS IoT ecosystem.
            """);
        waitForInputToContinue(scanner);
        iotActions.updateShadowThing(thingName);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Return a unique endpoint specific to the Amazon
Web Services account.");
        System.out.println("""
            An IoT Endpoint refers to a specific URL or Uniform Resource Locator
that serves as the entry point for communication between IoT devices and the AWS
IoT service.
           """);
        waitForInputToContinue(scanner);
        String endpointUrl = iotActions.describeEndpoint();
        System.out.println("The endpoint is "+endpointUrl);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. List your AWS IoT certificates");
        waitForInputToContinue(scanner);
        if (certificateArn.length() > 0) {
            iotActions.listCertificates();
        } else {
            System.out.println("You did not create a certificates. Skipping this
step.");
        }
        waitForInputToContinue(scanner);
        System.out.println(DASHES);
```

```
        System.out.println(DASHES);
        System.out.println("6. Create an IoT shadow that refers to a digital 
representation or virtual twin of a physical IoT device");
        System.out.println("""
            A Thing Shadow refers to a feature that enables you to create a 
virtual representation, or "shadow," 
            of a physical device or thing. The Thing Shadow allows you to 
synchronize and control the state of a device between 
            the cloud and the device itself. and the AWS IoT service. For 
example, you can write and retrieve JSON data from a Thing Shadow.
           """);
        waitForInputToContinue(scanner);
        iotActions.updateShadowThing(thingName);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Write out the state information, in JSON 
format.");
        waitForInputToContinue(scanner);
        iotActions.getPayload(thingName);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("8. Creates a rule");
        System.out.println("""
        Creates a rule that is an administrator-level action.
        Any user who has permission to create rules will be able to access data 
processed by the rule.
        """);
        System.out.print("Enter Rule name: ");
        ruleName = scanner.nextLine();
        iotActions.createIoTRule(roleARN, ruleName, snsAction);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("9. List your rules.");
        waitForInputToContinue(scanner);
        iotActions.listIoTRules();
        waitForInputToContinue(scanner);
        System.out.println(DASHES);
```

```
        System.out.println(DASHES);
        System.out.println("10. Search things using the Thing name.");
        waitForInputToContinue(scanner);
        String queryString = "thingName:"+thingName ;
        iotActions.searchThings(queryString);
        waitForInputToContinue(scanner);
        System.out.println(DASHES);

        System.out.println(DASHES);
        if (certificateArn.length() > 0) {
            System.out.print("Do you want to detach and delete the certificate 
for " +thingName +"? (y/n)");
            String delAns = scanner.nextLine();
            if (delAns != null && delAns.trim().equalsIgnoreCase("y")) {
                System.out.println("11. You selected to detach amd delete the 
certificate.");
                waitForInputToContinue(scanner);
                iotActions.detachThingPrincipal(thingName, certificateArn);
                iotActions.deleteCertificate(certificateArn);
                waitForInputToContinue(scanner);
            } else {
                System.out.println("11. You selected not to delete the 
certificate.");
            }
        } else {
            System.out.println("11. You did not create a certificate so there is 
nothing to delete.");
        }
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("12. Delete the AWS IoT Thing.");
        System.out.print("Do you want to delete the IoT Thing? (y/n)");
        String delAns = scanner.nextLine();
        if (delAns != null && delAns.trim().equalsIgnoreCase("y")) {
            iotActions.deleteIoTThing(thingName);
        } else {
            System.out.println("The IoT Thing was not deleted.");
        }
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS IoT workflow has successfully completed.");
```

```
        System.out.println(DASHES);
    }


    private static void waitForInputToContinue(Scanner scanner) {
        while (true) {
            System.out.println("");
            System.out.println("Enter 'c' followed by <ENTER> to continue:");
            String input = scanner.nextLine();

            if (input.trim().equalsIgnoreCase("c")) {
                System.out.println("Continuing with the program...");
                System.out.println("");
                break;
            } else {
                // Handle invalid input.
                System.out.println("Invalid input. Please try again.");
            }
        }
    }
}
```

A wrapper class for AWS IoT SDK methods.

```
import
 software.amazon.awssdk.auth.credentials.EnvironmentVariableCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryPolicy;
import software.amazon.awssdk.http.async.SdkAsyncHttpClient;
import software.amazon.awssdk.http.nio.netty.NettyNioAsyncHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iot.IotAsyncClient;
import software.amazon.awssdk.services.iot.model.Action;
import software.amazon.awssdk.services.iot.model.AttachThingPrincipalRequest;
import software.amazon.awssdk.services.iot.model.AttachThingPrincipalResponse;
import software.amazon.awssdk.services.iot.model.Certificate;
import
 software.amazon.awssdk.services.iot.model.CreateKeysAndCertificateResponse;
import software.amazon.awssdk.services.iot.model.CreateThingRequest;
import software.amazon.awssdk.services.iot.model.CreateThingResponse;
import software.amazon.awssdk.services.iot.model.CreateTopicRuleRequest;
```

```
import software.amazon.awssdk.services.iot.model.CreateTopicRuleResponse;
import software.amazon.awssdk.services.iot.model.DeleteCertificateRequest;
import software.amazon.awssdk.services.iot.model.DeleteCertificateResponse;
import software.amazon.awssdk.services.iot.model.DeleteThingRequest;
import software.amazon.awssdk.services.iot.model.DeleteThingResponse;
import software.amazon.awssdk.services.iot.model.DescribeEndpointRequest;
import software.amazon.awssdk.services.iot.model.DescribeEndpointResponse;
import software.amazon.awssdk.services.iot.model.DescribeThingRequest;
import software.amazon.awssdk.services.iot.model.DescribeThingResponse;
import software.amazon.awssdk.services.iot.model.DetachThingPrincipalRequest;
import software.amazon.awssdk.services.iot.model.DetachThingPrincipalResponse;
import software.amazon.awssdk.services.iot.model.IotException;
import software.amazon.awssdk.services.iot.model.ListCertificatesResponse;
import software.amazon.awssdk.services.iot.model.ListTopicRulesRequest;
import software.amazon.awssdk.services.iot.model.ListTopicRulesResponse;
import software.amazon.awssdk.services.iot.model.SearchIndexRequest;
import software.amazon.awssdk.services.iot.model.SearchIndexResponse;
import software.amazon.awssdk.services.iot.model.TopicRuleListItem;
import software.amazon.awssdk.services.iot.model.SnsAction;
import software.amazon.awssdk.services.iot.model.TopicRulePayload;
import software.amazon.awssdk.services.iotdataplane.IotDataPlaneAsyncClient;
import software.amazon.awssdk.services.iotdataplane.model.GetThingShadowRequest;
import software.amazon.awssdk.services.iotdataplane.model.GetThingShadowResponse;
import
 software.amazon.awssdk.services.iotdataplane.model.UpdateThingShadowRequest;
import
 software.amazon.awssdk.services.iotdataplane.model.UpdateThingShadowResponse;
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class IotActions {

    private static IotAsyncClient iotAsyncClient;

    private static IotDataPlaneAsyncClient iotAsyncDataPlaneClient;

    private static final String TOPIC = "your-iot-topic";

    private static IotDataPlaneAsyncClient getAsyncDataPlaneClient() {
```

```
        SdkAsyncHttpClient httpClient = NettyNioAsyncHttpClient.builder()
            .maxConcurrency(100)
            .connectionTimeout(Duration.ofSeconds(60))
            .readTimeout(Duration.ofSeconds(60))
            .writeTimeout(Duration.ofSeconds(60))
            .build();

        ClientOverrideConfiguration overrideConfig =
 ClientOverrideConfiguration.builder()
            .apiCallTimeout(Duration.ofMinutes(2))
            .apiCallAttemptTimeout(Duration.ofSeconds(90))
            .retryPolicy(RetryPolicy.builder()
                .numRetries(3)
                .build())
            .build();

        if (iotAsyncDataPlaneClient == null) {
            iotAsyncDataPlaneClient = IotDataPlaneAsyncClient.builder()
                .region(Region.US_EAST_1)
                .httpClient(httpClient)
                .overrideConfiguration(overrideConfig)
                .build();
        }
        return iotAsyncDataPlaneClient;
    }


    private static IotAsyncClient getAsyncClient() {
        SdkAsyncHttpClient httpClient = NettyNioAsyncHttpClient.builder()
            .maxConcurrency(100)
            .connectionTimeout(Duration.ofSeconds(60))
            .readTimeout(Duration.ofSeconds(60))
            .writeTimeout(Duration.ofSeconds(60))
            .build();

        ClientOverrideConfiguration overrideConfig =
 ClientOverrideConfiguration.builder()
            .apiCallTimeout(Duration.ofMinutes(2))
            .apiCallAttemptTimeout(Duration.ofSeconds(90))
            .retryPolicy(RetryPolicy.builder()
                .numRetries(3)
                .build())
            .build();
```

```
        if (iotAsyncClient == null) {
            iotAsyncClient = IotAsyncClient.builder()
                .region(Region.US_EAST_1)
                .httpClient(httpClient)
                .overrideConfiguration(overrideConfig)
                .build();
        }
        return iotAsyncClient;
    }

    /**
     * Creates an IoT certificate asynchronously.
     *
     * @return The ARN of the created certificate.
     * <p>
     * This method initiates an asynchronous request to create an IoT
certificate.
     * If the request is successful, it prints the certificate details and
returns the certificate ARN.
     * If an exception occurs, it prints the error message.
     */
    public String createCertificate() {
        CompletableFuture<CreateKeysAndCertificateResponse> future =
getAsyncClient().createKeysAndCertificate();
        final String[] certificateArn = {null};
        future.whenComplete((response, ex) -> {
            if (response != null) {
                String certificatePem = response.certificatePem();
                certificateArn[0] = response.certificateArn();

                // Print the details.
                System.out.println("\nCertificate:");
                System.out.println(certificatePem);
                System.out.println("\nCertificate ARN:");
                System.out.println(certificateArn[0]);

            } else {
                Throwable cause = (ex instanceof CompletionException) ?
ex.getCause() : ex;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else {
```

```
                    System.err.println("Unexpected error: " +
cause.getMessage());
                }
            }
        });

        future.join();
        return certificateArn[0];
    }

    /**
     * Creates an IoT Thing with the specified name asynchronously.
     *
     * @param thingName The name of the IoT Thing to create.
     *
     * This method initiates an asynchronous request to create an IoT Thing with
the specified name.
     * If the request is successful, it prints the name of the thing and its ARN
value.
     * If an exception occurs, it prints the error message.
     */
    public void createIoTThing(String thingName) {
        CreateThingRequest createThingRequest = CreateThingRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<CreateThingResponse> future =
getAsyncClient().createThing(createThingRequest);
        future.whenComplete((createThingResponse, ex) -> {
            if (createThingResponse != null) {
                System.out.println(thingName + " was successfully created. The
ARN value is " + createThingResponse.thingArn());
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                }
            }
        });
```

```
        future.join();
    }

    /**
     * Attaches a certificate to an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     * @param certificateArn The ARN of the certificate to attach.
     *
     * This method initiates an asynchronous request to attach a certificate to
 an IoT Thing.
     * If the request is successful, it prints a confirmation message and
 additional information about the Thing.
     * If an exception occurs, it prints the error message.
     */
    public void attachCertificateToThing(String thingName, String certificateArn)
 {
        AttachThingPrincipalRequest principalRequest =
 AttachThingPrincipalRequest.builder()
            .thingName(thingName)
            .principal(certificateArn)
            .build();

        CompletableFuture<AttachThingPrincipalResponse> future =
 getAsyncClient().attachThingPrincipal(principalRequest);
        future.whenComplete((attachResponse, ex) -> {
            if (attachResponse != null &&
 attachResponse.sdkHttpResponse().isSuccessful()) {
                System.out.println("Certificate attached to Thing
 successfully.");

                // Print additional information about the Thing.
                describeThing(thingName);
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                } else {
                    System.err.println("Failed to attach certificate to Thing.
 HTTP Status Code: " +
```

```
                        attachResponse.sdkHttpResponse().statusCode());
                }
            }
        });

        future.join();
    }

    /**
     * Describes an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     *
     * This method initiates an asynchronous request to describe an IoT Thing.
     * If the request is successful, it prints the Thing details.
     * If an exception occurs, it prints the error message.
     */
    private void describeThing(String thingName) {
        DescribeThingRequest thingRequest = DescribeThingRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<DescribeThingResponse> future =
getAsyncClient().describeThing(thingRequest);
        future.whenComplete((describeResponse, ex) -> {
            if (describeResponse != null) {
                System.out.println("Thing Details:");
                System.out.println("Thing Name: " +
describeResponse.thingName());
                System.out.println("Thing ARN: " + describeResponse.thingArn());
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to describe Thing.");
                }
            }
        });
```

```
        future.join();
    }

    /**
     * Updates the shadow of an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     *
     * This method initiates an asynchronous request to update the shadow of an
 IoT Thing.
     * If the request is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void updateShadowThing(String thingName) {
        // Create Thing Shadow State Document.
        String stateDocument = "{\"state\":{\"reported\":{\"temperature\":25,
 \"humidity\":50}}}";
        SdkBytes data = SdkBytes.fromString(stateDocument,
 StandardCharsets.UTF_8);
        UpdateThingShadowRequest updateThingShadowRequest =
 UpdateThingShadowRequest.builder()
            .thingName(thingName)
            .payload(data)
            .build();

        CompletableFuture<UpdateThingShadowResponse> future =
 getAsyncDataPlaneClient().updateThingShadow(updateThingShadowRequest);
        future.whenComplete((updateResponse, ex) -> {
            if (updateResponse != null) {
                System.out.println("Thing Shadow updated successfully.");
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                } else {
                    System.err.println("Failed to update Thing Shadow.");
                }
            }
        });
```

```
        future.join();
    }

    /**
     * Describes the endpoint of the IoT service asynchronously.
     *
     * @return A CompletableFuture containing the full endpoint URL.
     *
     * This method initiates an asynchronous request to describe the endpoint of
 the IoT service.
     * If the request is successful, it prints and returns the full endpoint URL.
     * If an exception occurs, it prints the error message.
     */
    public String describeEndpoint() {
        CompletableFuture<DescribeEndpointResponse> future =
 getAsyncClient().describeEndpoint(DescribeEndpointRequest.builder().endpointType("iot:Da
ATS").build());
        final String[] result = {null};

        future.whenComplete((endpointResponse, ex) -> {
            if (endpointResponse != null) {
                String endpointUrl = endpointResponse.endpointAddress();
                String exString = getValue(endpointUrl);
                String fullEndpoint = "https://" + exString + "-ats.iot.us-
east-1.amazonaws.com";

                System.out.println("Full Endpoint URL: " + fullEndpoint);
                result[0] = fullEndpoint;
            } else {
                Throwable cause = (ex instanceof CompletionException) ?
 ex.getCause() : ex;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                }
            }
        });

        future.join();
        return result[0];
    }
```

```
    /**
     * Extracts a specific value from the endpoint URL.
     *
     * @param input The endpoint URL to process.
     * @return The extracted value from the endpoint URL.
     */
    private static String getValue(String input) {
        // Define a regular expression pattern for extracting the subdomain.
        Pattern pattern = Pattern.compile("^(.*?)\\.iot\\.us-east-1\\.amazonaws\
\.com");

        // Match the pattern against the input string.
        Matcher matcher = pattern.matcher(input);

        // Check if a match is found.
        if (matcher.find()) {
            // Extract the subdomain from the first capturing group.
            String subdomain = matcher.group(1);
            System.out.println("Extracted subdomain: " + subdomain);
            return subdomain ;
        } else {
            System.out.println("No match found");
        }
        return "" ;
    }

    /**
     * Lists all certificates asynchronously.
     *
     * This method initiates an asynchronous request to list all certificates.
     * If the request is successful, it prints the certificate IDs and ARNs.
     * If an exception occurs, it prints the error message.
     */
    public void listCertificates() {
        CompletableFuture<ListCertificatesResponse> future =
 getAsyncClient().listCertificates();
        future.whenComplete((response, ex) -> {
            if (response != null) {
                List<Certificate> certList = response.certificates();
                for (Certificate cert : certList) {
                    System.out.println("Cert id: " + cert.certificateId());
                    System.out.println("Cert Arn: " + cert.certificateArn());
                }
```

```
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to list certificates.");
                }
            }
        });

        future.join();
    }

    /**
     * Retrieves the payload of a Thing's shadow asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     *
     * This method initiates an asynchronous request to get the payload of a
Thing's shadow.
     * If the request is successful, it prints the shadow data.
     * If an exception occurs, it prints the error message.
     */
    public void getPayload(String thingName) {
        GetThingShadowRequest getThingShadowRequest =
GetThingShadowRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<GetThingShadowResponse> future =
getAsyncDataPlaneClient().getThingShadow(getThingShadowRequest);
        future.whenComplete((getThingShadowResponse, ex) -> {
            if (getThingShadowResponse != null) {
                // Extracting payload from response.
                SdkBytes payload = getThingShadowResponse.payload();
                String payloadString = payload.asUtf8String();
                System.out.println("Received Shadow Data: " + payloadString);
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
```

```
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to get Thing Shadow payload.");
                }
            }
        });

        future.join();
    }

    /**
     * Creates an IoT rule asynchronously.
     *
     * @param roleARN The ARN of the IAM role that grants access to the rule's
actions.
     * @param ruleName The name of the IoT rule.
     * @param action The ARN of the action to perform when the rule is triggered.
     *
     * This method initiates an asynchronous request to create an IoT rule.
     * If the request is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void createIoTRule(String roleARN, String ruleName, String action) {
        String sql = "SELECT * FROM '" + TOPIC + "'";
        SnsAction action1 = SnsAction.builder()
            .targetArn(action)
            .roleArn(roleARN)
            .build();

        // Create the action.
        Action myAction = Action.builder()
            .sns(action1)
            .build();

        // Create the topic rule payload.
        TopicRulePayload topicRulePayload = TopicRulePayload.builder()
            .sql(sql)
            .actions(myAction)
            .build();
```

```
        // Create the topic rule request.
        CreateTopicRuleRequest topicRuleRequest =
CreateTopicRuleRequest.builder()
            .ruleName(ruleName)
            .topicRulePayload(topicRulePayload)
            .build();

        CompletableFuture<CreateTopicRuleResponse> future =
getAsyncClient().createTopicRule(topicRuleRequest);
        future.whenComplete((response, ex) -> {
            if (response != null) {
                System.out.println("IoT Rule created successfully.");
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to create IoT Rule.");
                }
            }
        });

        future.join();
    }

    /**
     * Lists IoT rules asynchronously.
     *
     * This method initiates an asynchronous request to list IoT rules.
     * If the request is successful, it prints the names and ARNs of the rules.
     * If an exception occurs, it prints the error message.
     */
    public void listIoTRules() {
        ListTopicRulesRequest listTopicRulesRequest =
ListTopicRulesRequest.builder().build();
        CompletableFuture<ListTopicRulesResponse> future =
getAsyncClient().listTopicRules(listTopicRulesRequest);
        future.whenComplete((listTopicRulesResponse, ex) -> {
            if (listTopicRulesResponse != null) {
                System.out.println("List of IoT Rules:");
```

```
                List<TopicRuleListItem> ruleList =
listTopicRulesResponse.rules();
                for (TopicRuleListItem rule : ruleList) {
                    System.out.println("Rule Name: " + rule.ruleName());
                    System.out.println("Rule ARN: " + rule.ruleArn());
                    System.out.println("--------------");
                }
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to list IoT Rules.");
                }
            }
        });

        future.join();
    }

    /**
     * Searches for IoT Things asynchronously based on a query string.
     *
     * @param queryString The query string to search for Things.
     *
     * This method initiates an asynchronous request to search for IoT Things.
     * If the request is successful and Things are found, it prints their IDs.
     * If no Things are found, it prints a message indicating so.
     * If an exception occurs, it prints the error message.
     */
    public void searchThings(String queryString) {
        SearchIndexRequest searchIndexRequest = SearchIndexRequest.builder()
            .queryString(queryString)
            .build();

        CompletableFuture<SearchIndexResponse> future =
getAsyncClient().searchIndex(searchIndexRequest);
        future.whenComplete((searchIndexResponse, ex) -> {
            if (searchIndexResponse != null) {
                // Process the result.
```

```
                if (searchIndexResponse.things().isEmpty()) {
                    System.out.println("No things found.");
                } else {
                    searchIndexResponse.things().forEach(thing ->
System.out.println("Thing id found using search is " + thing.thingId()));
                }
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to search for IoT Things.");
                }
            }
        });

        future.join();
    }

    /**
     * Detaches a principal (certificate) from an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     * @param certificateArn The ARN of the certificate to detach.
     *
     * This method initiates an asynchronous request to detach a certificate from
an IoT Thing.
     * If the detachment is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void detachThingPrincipal(String thingName, String certificateArn) {
        DetachThingPrincipalRequest thingPrincipalRequest =
DetachThingPrincipalRequest.builder()
            .principal(certificateArn)
            .thingName(thingName)
            .build();

        CompletableFuture<DetachThingPrincipalResponse> future =
getAsyncClient().detachThingPrincipal(thingPrincipalRequest);
        future.whenComplete((voidResult, ex) -> {
```

```
            if (ex == null) {
                System.out.println(certificateArn + " was successfully removed
 from " + thingName);
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }

    /**
     * Deletes a certificate asynchronously.
     *
     * @param certificateArn The ARN of the certificate to delete.
     *
     * This method initiates an asynchronous request to delete a certificate.
     * If the deletion is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void deleteCertificate(String certificateArn) {
        DeleteCertificateRequest certificateProviderRequest =
 DeleteCertificateRequest.builder()
            .certificateId(extractCertificateId(certificateArn))
            .build();

        CompletableFuture<DeleteCertificateResponse> future =
 getAsyncClient().deleteCertificate(certificateProviderRequest);
        future.whenComplete((voidResult, ex) -> {
            if (ex == null) {
                System.out.println(certificateArn + " was successfully
 deleted.");
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
```

```
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }

    /**
     * Deletes an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing to delete.
     *
     * This method initiates an asynchronous request to delete an IoT Thing.
     * If the deletion is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void deleteIoTThing(String thingName) {
        DeleteThingRequest deleteThingRequest = DeleteThingRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<DeleteThingResponse> future =
 getAsyncClient().deleteThing(deleteThingRequest);
        future.whenComplete((voidResult, ex) -> {
            if (ex == null) {
                System.out.println("Deleted Thing " + thingName);
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }

    // Get the cert Id  from the Cert ARN value.
    private String extractCertificateId(String certificateArn) {
        // Example ARN: arn:aws:iot:region:account-id:cert/certificate-id.
```

```
        String[] arnParts = certificateArn.split(":");
        String certificateIdPart = arnParts[arnParts.length - 1];
        return certificateIdPart.substring(certificateIdPart.lastIndexOf("/") +
 1);
    }
}
```

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/kotlin/services/iot#code-examples).

```
import aws.sdk.kotlin.services.iot.IotClient
import aws.sdk.kotlin.services.iot.model.Action
import aws.sdk.kotlin.services.iot.model.AttachThingPrincipalRequest
import aws.sdk.kotlin.services.iot.model.AttributePayload
import aws.sdk.kotlin.services.iot.model.CreateThingRequest
import aws.sdk.kotlin.services.iot.model.CreateTopicRuleRequest
import aws.sdk.kotlin.services.iot.model.DeleteCertificateRequest
import aws.sdk.kotlin.services.iot.model.DeleteThingRequest
import aws.sdk.kotlin.services.iot.model.DescribeEndpointRequest
import aws.sdk.kotlin.services.iot.model.DescribeThingRequest
import aws.sdk.kotlin.services.iot.model.DetachThingPrincipalRequest
import aws.sdk.kotlin.services.iot.model.ListTopicRulesRequest
import aws.sdk.kotlin.services.iot.model.SearchIndexRequest
import aws.sdk.kotlin.services.iot.model.SnsAction
import aws.sdk.kotlin.services.iot.model.TopicRulePayload
import aws.sdk.kotlin.services.iot.model.UpdateThingRequest
import aws.sdk.kotlin.services.iotdataplane.IotDataPlaneClient
import aws.sdk.kotlin.services.iotdataplane.model.GetThingShadowRequest
import aws.sdk.kotlin.services.iotdataplane.model.UpdateThingShadowRequest
import aws.smithy.kotlin.runtime.content.ByteStream
import aws.smithy.kotlin.runtime.content.toByteArray
import java.util.Scanner
import java.util.regex.Pattern
```

```
import kotlin.system.exitProcess

/**
 * Before running this Kotlin code example, ensure that your development
 environment
 * is set up, including configuring your credentials.
 *
 * For detailed instructions, refer to the following documentation topic:
 * [Setting Up Your Development Environment](https://docs.aws.amazon.com/sdk-for-
kotlin/latest/developer-guide/setup.html)
 *
 * This code example requires an SNS topic and an IAM Role.
 * Follow the steps in the documentation to set up these resources:
 *
 * - [Creating an SNS Topic](https://docs.aws.amazon.com/sns/latest/dg/sns-
getting-started.html#step-create-topic)
 * - [Creating an IAM Role](https://docs.aws.amazon.com/IAM/latest/UserGuide/
id_roles_create.html)
 */

val DASHES = String(CharArray(80)).replace("\u0000", "-")
val TOPIC = "your-iot-topic"

suspend fun main(args: Array<String>) {
    val usage =
        """
        Usage:
            <roleARN> <snsAction>

        Where:
            roleARN - The ARN of an IAM role that has permission to work with AWS
 IOT.
            snsAction  - An ARN of an SNS topic.

        """.trimIndent()

    if (args.size != 2) {
        println(usage)
        exitProcess(1)
    }

    var thingName: String
    val roleARN = args[0]
    val snsAction = args[1]
```

```
    val scanner = Scanner(System.`in`)

    println(DASHES)
    println("Welcome to the AWS IoT example scenario.")
    println(
        """
        This example program demonstrates various interactions with the AWS
 Internet of Things (IoT) Core service.
        The program guides you through a series of steps, including creating an
 IoT thing, generating a device certificate,
        updating the thing with attributes, and so on.

        It utilizes the AWS SDK for Kotlin and incorporates functionality for
 creating and managing IoT things, certificates, rules,
        shadows, and performing searches. The program aims to showcase AWS IoT
 capabilities and provides a comprehensive example for
        developers working with AWS IoT in a Kotlin environment.
        """.trimIndent(),
    )

    print("Press Enter to continue...")
    scanner.nextLine()
    println(DASHES)

    println(DASHES)
    println("1. Create an AWS IoT thing.")
    println(
        """
        An AWS IoT thing represents a virtual entity in the AWS IoT service that
 can be associated with a physical device.
        """.trimIndent(),
    )
    // Prompt the user for input.
    print("Enter thing name: ")
    thingName = scanner.nextLine()
    createIoTThing(thingName)
    describeThing(thingName)
    println(DASHES)

    println(DASHES)
    println("2. Generate a device certificate.")
    println(
        """
```

```
        A device certificate performs a role in securing the communication
between devices (things) and the AWS IoT platform.
        """.trimIndent(),
    )

    print("Do you want to create a certificate for $thingName? (y/n)")
    val certAns = scanner.nextLine()
    var certificateArn: String? = ""
    if (certAns != null && certAns.trim { it <= ' ' }.equals("y", ignoreCase =
true)) {
        certificateArn = createCertificate()
        println("Attach the certificate to the AWS IoT thing.")
        attachCertificateToThing(thingName, certificateArn)
    } else {
        println("A device certificate was not created.")
    }
    println(DASHES)

    println(DASHES)
    println("3. Update an AWS IoT thing with Attributes.")
    println(
        """
        IoT thing attributes, represented as key-value pairs, offer a pivotal
advantage in facilitating efficient data
        management and retrieval within the AWS IoT ecosystem.
        """.trimIndent(),
    )
    print("Press Enter to continue...")
    scanner.nextLine()
    updateThing(thingName)
    println(DASHES)

    println(DASHES)
    println("4. Return a unique endpoint specific to the Amazon Web Services
account.")
    println(
        """
        An IoT Endpoint refers to a specific URL or Uniform Resource Locator that
serves as the entry point for communication between IoT devices and the AWS IoT
service.
        """.trimIndent(),
    )
    print("Press Enter to continue...")
    scanner.nextLine()
```

```
    val endpointUrl = describeEndpoint()
    println(DASHES)

    println(DASHES)
    println("5. List your AWS IoT certificates")
    print("Press Enter to continue...")
    scanner.nextLine()
    if (certificateArn!!.isNotEmpty()) {
        listCertificates()
    } else {
        println("You did not create a certificates. Skipping this step.")
    }
    println(DASHES)

    println(DASHES)
    println("6. Create an IoT shadow that refers to a digital representation or
 virtual twin of a physical IoT device")
    println(
        """
        A thing shadow refers to a feature that enables you to create a virtual
 representation, or "shadow,"
        of a physical device or thing. The thing shadow allows you to synchronize
 and control the state of a device between
        the cloud and the device itself. and the AWS IoT service. For example,
 you can write and retrieve JSON data from a thing shadow.

        """.trimIndent(),
    )
    print("Press Enter to continue...")
    scanner.nextLine()
    updateShawdowThing(thingName)
    println(DASHES)

    println(DASHES)
    println("7. Write out the state information, in JSON format.")
    print("Press Enter to continue...")
    scanner.nextLine()
    getPayload(thingName)
    println(DASHES)

    println(DASHES)
    println("8. Creates a rule")
    println(
        """
```

```
        Creates a rule that is an administrator-level action.
        Any user who has permission to create rules will be able to access data 
processed by the rule.
        """.trimIndent(),
    )
    print("Enter Rule name: ")
    val ruleName = scanner.nextLine()
    createIoTRule(roleARN, ruleName, snsAction)
    println(DASHES)

    println(DASHES)
    println("9. List your rules.")
    print("Press Enter to continue...")
    scanner.nextLine()
    listIoTRules()
    println(DASHES)

    println(DASHES)
    println("10. Search things using the name.")
    print("Press Enter to continue...")
    scanner.nextLine()
    val queryString = "thingName:$thingName"
    searchThings(queryString)
    println(DASHES)

    println(DASHES)
    if (certificateArn.length > 0) {
        print("Do you want to detach and delete the certificate for $thingName? 
(y/n)")
        val delAns = scanner.nextLine()
        if (delAns != null && delAns.trim { it <= ' ' }.equals("y", ignoreCase = 
true)) {
            println("11. You selected to detach amd delete the certificate.")
            print("Press Enter to continue...")
            scanner.nextLine()
            detachThingPrincipal(thingName, certificateArn)
            deleteCertificate(certificateArn)
        } else {
            println("11. You selected not to delete the certificate.")
        }
    } else {
        println("11. You did not create a certificate so there is nothing to 
delete.")
    }
```

```
    println(DASHES)

    println(DASHES)
    println("12. Delete the AWS IoT thing.")
    print("Do you want to delete the IoT thing? (y/n)")
    val delAns = scanner.nextLine()
    if (delAns != null && delAns.trim { it <= ' ' }.equals("y", ignoreCase =
 true)) {
        deleteIoTThing(thingName)
    } else {
        println("The IoT thing was not deleted.")
    }
    println(DASHES)

    println(DASHES)
    println("The AWS IoT workflow has successfully completed.")
    println(DASHES)
}

suspend fun deleteIoTThing(thingNameVal: String) {
    val deleteThingRequest =
        DeleteThingRequest {
            thingName = thingNameVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.deleteThing(deleteThingRequest)
        println("Deleted $thingNameVal")
    }
}

suspend fun deleteCertificate(certificateArn: String) {
    val certificateProviderRequest =
        DeleteCertificateRequest {
            certificateId = extractCertificateId(certificateArn)
        }
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.deleteCertificate(certificateProviderRequest)
        println("$certificateArn was successfully deleted.")
    }
}

private fun extractCertificateId(certificateArn: String): String? {
    // Example ARN: arn:aws:iot:region:account-id:cert/certificate-id.
```

```
    val arnParts = certificateArn.split(":".toRegex()).dropLastWhile
 { it.isEmpty() }.toTypedArray()
    val certificateIdPart = arnParts[arnParts.size - 1]
    return certificateIdPart.substring(certificateIdPart.lastIndexOf("/") + 1)
}

suspend fun detachThingPrincipal(
    thingNameVal: String,
    certificateArn: String,
) {
    val thingPrincipalRequest =
        DetachThingPrincipalRequest {
            principal = certificateArn
            thingName = thingNameVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.detachThingPrincipal(thingPrincipalRequest)
        println("$certificateArn was successfully removed from $thingNameVal")
    }
}

suspend fun searchThings(queryStringVal: String?) {
    val searchIndexRequest =
        SearchIndexRequest {
            queryString = queryStringVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val searchIndexResponse = iotClient.searchIndex(searchIndexRequest)
        if (searchIndexResponse.things?.isEmpty() == true) {
            println("No things found.")
        } else {
            searchIndexResponse.things
                ?.forEach { thing -> println("Thing id found using search is
 ${thing.thingId}") }
        }
    }
}

suspend fun listIoTRules() {
    val listTopicRulesRequest = ListTopicRulesRequest {}

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
```

```
        val listTopicRulesResponse =
 iotClient.listTopicRules(listTopicRulesRequest)
        println("List of IoT rules:")
        val ruleList = listTopicRulesResponse.rules
        ruleList?.forEach { rule ->
            println("Rule name: ${rule.ruleName}")
            println("Rule ARN: ${rule.ruleArn}")
            println("--------------")
        }
    }
}

suspend fun createIoTRule(
    roleARNVal: String?,
    ruleNameVal: String?,
    action: String?,
) {
    val sqlVal = "SELECT * FROM '$TOPIC '"
    val action1 =
        SnsAction {
            targetArn = action
            roleArn = roleARNVal
        }

    val myAction =
        Action {
            sns = action1
        }

    val topicRulePayloadVal =
        TopicRulePayload {
            sql = sqlVal
            actions = listOf(myAction)
        }

    val topicRuleRequest =
        CreateTopicRuleRequest {
            ruleName = ruleNameVal
            topicRulePayload = topicRulePayloadVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.createTopicRule(topicRuleRequest)
        println("IoT rule created successfully.")
```

```
    }
}

suspend fun getPayload(thingNameVal: String?) {
    val getThingShadowRequest =
        GetThingShadowRequest {
            thingName = thingNameVal
        }

    IotDataPlaneClient.fromEnvironment { region = "us-east-1" }.use
 { iotPlaneClient ->
        val getThingShadowResponse =
 iotPlaneClient.getThingShadow(getThingShadowRequest)
        val payload = getThingShadowResponse.payload
        val payloadString = payload?.let { java.lang.String(it, Charsets.UTF_8) }
        println("Received shadow data: $payloadString")
    }
}

suspend fun listCertificates() {
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val response = iotClient.listCertificates()
        val certList = response.certificates
        certList?.forEach { cert ->
            println("Cert id: ${cert.certificateId}")
            println("Cert Arn: ${cert.certificateArn}")
        }
    }
}

suspend fun describeEndpoint(): String? {
    val request = DescribeEndpointRequest {}
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val endpointResponse = iotClient.describeEndpoint(request)
        val endpointUrl: String? = endpointResponse.endpointAddress
        val exString: String = getValue(endpointUrl)
        val fullEndpoint = "https://$exString-ats.iot.us-east-1.amazonaws.com"
        println("Full endpoint URL: $fullEndpoint")
        return fullEndpoint
    }
}

private fun getValue(input: String?): String {
    // Define a regular expression pattern for extracting the subdomain.
```

```
    val pattern = Pattern.compile("^(.*?)\\.iot\\.us-east-1\\.amazonaws\\.com")

    // Match the pattern against the input string.
    val matcher = pattern.matcher(input)

    // Check if a match is found.
    if (matcher.find()) {
        val subdomain = matcher.group(1)
        println("Extracted subdomain: $subdomain")
        return subdomain
    } else {
        println("No match found")
    }
    return ""
}

suspend fun updateThing(thingNameVal: String?) {
    val newLocation = "Office"
    val newFirmwareVersion = "v2.0"
    val attMap: MutableMap<String, String> = HashMap()
    attMap["location"] = newLocation
    attMap["firmwareVersion"] = newFirmwareVersion

    val attributePayloadVal =
        AttributePayload {
            attributes = attMap
        }

    val updateThingRequest =
        UpdateThingRequest {
            thingName = thingNameVal
            attributePayload = attributePayloadVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        // Update the IoT thing attributes.
        iotClient.updateThing(updateThingRequest)
        println("$thingNameVal attributes updated successfully.")
    }
}

suspend fun updateShawdowThing(thingNameVal: String?) {
    // Create the thing shadow state document.
```

```
    val stateDocument = "{\"state\":{\"reported\":{\"temperature\":25, \"humidity
\":50}}}"
    val byteStream: ByteStream = ByteStream.fromString(stateDocument)
    val byteArray: ByteArray = byteStream.toByteArray()

    val updateThingShadowRequest =
        UpdateThingShadowRequest {
            thingName = thingNameVal
            payload = byteArray
        }

    IotDataPlaneClient.fromEnvironment { region = "us-east-1" }.use
 { iotPlaneClient ->
        iotPlaneClient.updateThingShadow(updateThingShadowRequest)
        println("The thing shadow was updated successfully.")
    }
}

suspend fun attachCertificateToThing(
    thingNameVal: String?,
    certificateArn: String?,
) {
    val principalRequest =
        AttachThingPrincipalRequest {
            thingName = thingNameVal
            principal = certificateArn
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.attachThingPrincipal(principalRequest)
        println("Certificate attached to $thingNameVal successfully.")
    }
}

suspend fun describeThing(thingNameVal: String) {
    val thingRequest =
        DescribeThingRequest {
            thingName = thingNameVal
        }

    // Print Thing details.
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val describeResponse = iotClient.describeThing(thingRequest)
        println("Thing details:")
```

```
        println("Thing name: ${describeResponse.thingName}")
        println("Thing ARN:  ${describeResponse.thingArn}")
    }
}

suspend fun createCertificate(): String? {
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val response = iotClient.createKeysAndCertificate()
        val certificatePem = response.certificatePem
        val certificateArn = response.certificateArn

        // Print the details.
        println("\nCertificate:")
        println(certificatePem)
        println("\nCertificate ARN:")
        println(certificateArn)
        return certificateArn
    }
}

suspend fun createIoTThing(thingNameVal: String) {
    val createThingRequest =
        CreateThingRequest {
            thingName = thingNameVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.createThing(createThingRequest)
        println("Created $thingNameVal}")
    }
}
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

Create an IoT wrapper class to manage operations.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)


    def create_thing(self, thing_name):
        """
        Creates an AWS IoT thing.

        :param thing_name: The name of the thing to create.
        :return: The name and ARN of the created thing.
        """
        try:
            response = self.iot_client.create_thing(thingName=thing_name)
            logger.info("Created thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceAlreadyExistsException":
                logger.info("Thing %s already exists. Skipping creation.",
 thing_name)
                return None
            logger.error(
                "Couldn't create thing %s. Here's why: %s: %s",
                thing_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

```
    def list_things(self):
        """
        Lists AWS IoT things.

        :return: The list of things.
        """
        try:
            things = []
            paginator = self.iot_client.get_paginator("list_things")
            for page in paginator.paginate():
                things.extend(page["things"])
            logger.info("Retrieved %s things.", len(things))
            return things
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't list things. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise



    def create_keys_and_certificate(self):
        """
        Creates keys and a certificate for an AWS IoT thing.

        :return: The certificate ID, ARN, and PEM.
        """
        try:
            response =
 self.iot_client.create_keys_and_certificate(setAsActive=True)
            logger.info("Created certificate %s.", response["certificateId"])
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't create keys and certificate. Here's why: %s: %s",
```

```
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response


    def attach_thing_principal(self, thing_name, principal):
        """
        Attaches a certificate to an AWS IoT thing.

        :param thing_name: The name of the thing.
        :param principal: The ARN of the certificate.
        """
        try:
            self.iot_client.attach_thing_principal(
                thingName=thing_name, principal=principal
            )
            logger.info("Attached principal %s to thing %s.", principal,
 thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot attach principal. Resource not found.")
                return
            logger.error(
                "Couldn't attach principal to thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def describe_endpoint(self, endpoint_type="iot:Data-ATS"):
        """
        Gets the AWS IoT endpoint.

        :param endpoint_type: The endpoint type.
        :return: The endpoint.
        """
        try:
            response =
 self.iot_client.describe_endpoint(endpointType=endpoint_type)
            logger.info("Retrieved endpoint %s.", response["endpointAddress"])
```

```
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't describe endpoint. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response["endpointAddress"]


    def list_certificates(self):
        """
        Lists AWS IoT certificates.

        :return: The list of certificates.
        """
        try:
            certificates = []
            paginator = self.iot_client.get_paginator("list_certificates")
            for page in paginator.paginate():
                certificates.extend(page["certificates"])
            logger.info("Retrieved %s certificates.", len(certificates))
            return certificates
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't list certificates. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise


    def detach_thing_principal(self, thing_name, principal):
        """
        Detaches a certificate from an AWS IoT thing.

        :param thing_name: The name of the thing.
```

```
        :param principal: The ARN of the certificate.
        """
        try:
            self.iot_client.detach_thing_principal(
                thingName=thing_name, principal=principal
            )
            logger.info("Detached principal %s from thing %s.", principal,
 thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot detach principal. Resource not found.")
                return
            logger.error(
                "Couldn't detach principal from thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def delete_certificate(self, certificate_id):
        """
        Deletes an AWS IoT certificate.

        :param certificate_id: The ID of the certificate to delete.
        """
        try:
            self.iot_client.update_certificate(
                certificateId=certificate_id, newStatus="INACTIVE"
            )
            self.iot_client.delete_certificate(certificateId=certificate_id)
            logger.info("Deleted certificate %s.", certificate_id)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot delete certificate. Resource not found.")
                return
            logger.error(
                "Couldn't delete certificate. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

```
    def create_topic_rule(self, rule_name, topic, sns_action_arn, role_arn):
        """
        Creates an AWS IoT topic rule.

        :param rule_name: The name of the rule.
        :param topic: The MQTT topic to subscribe to.
        :param sns_action_arn: The ARN of the SNS topic to publish to.
        :param role_arn: The ARN of the IAM role.
        """
        try:
            self.iot_client.create_topic_rule(
                ruleName=rule_name,
                topicRulePayload={
                    "sql": f"SELECT * FROM '{topic}'",
                    "actions": [
                        {"sns": {"targetArn": sns_action_arn, "roleArn":
 role_arn}}
                    ],
                },
            )
            logger.info("Created topic rule %s.", rule_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceAlreadyExistsException":
                logger.info("Topic rule %s already exists. Skipping creation.",
 rule_name)
                return
            logger.error(
                "Couldn't create topic rule. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def list_topic_rules(self):
        """
        Lists AWS IoT topic rules.

        :return: The list of topic rules.
        """
        try:
            rules = []
            paginator = self.iot_client.get_paginator("list_topic_rules")
            for page in paginator.paginate():
```

```
                rules.extend(page["rules"])
            logger.info("Retrieved %s topic rules.", len(rules))
            return rules
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't list topic rules. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise



    def search_index(self, query):
        """
        Searches the AWS IoT index.

        :param query: The search query.
        :return: The list of things found.
        """
        try:
            response = self.iot_client.search_index(queryString=query)
            logger.info("Found %s things.", len(response.get("things", [])))
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't search index. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response.get("things", [])


    def update_indexing_configuration(self):
        """
        Updates the AWS IoT indexing configuration to enable thing indexing.
        """
```

```
        try:
            self.iot_client.update_indexing_configuration(
                thingIndexingConfiguration={"thingIndexingMode": "REGISTRY"}
            )
            logger.info("Updated indexing configuration.")
        except ClientError as err:
            logger.error(
                "Couldn't update indexing configuration. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def delete_thing(self, thing_name):
        """
        Deletes an AWS IoT thing.

        :param thing_name: The name of the thing to delete.
        """
        try:
            self.iot_client.delete_thing(thingName=thing_name)
            logger.info("Deleted thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot delete thing. Resource not found.")
                return
            logger.error(
                "Couldn't delete thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def delete_topic_rule(self, rule_name):
        """
        Deletes an AWS IoT topic rule.

        :param rule_name: The name of the rule to delete.
        """
        try:
            self.iot_client.delete_topic_rule(ruleName=rule_name)
            logger.info("Deleted topic rule %s.", rule_name)
```

```
        except ClientError as err:
            logger.error(
                "Couldn't delete topic rule. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def update_thing_shadow(self, thing_name, shadow_state):
        """
        Updates the shadow for an AWS IoT thing.

        :param thing_name: The name of the thing.
        :param shadow_state: The shadow state as a dictionary.
        """
        import json
        try:
            self.iot_data_client.update_thing_shadow(
                thingName=thing_name, payload=json.dumps(shadow_state)
            )
            logger.info("Updated shadow for thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot update thing shadow. Resource not found.")
                return
            logger.error(
                "Couldn't update thing shadow. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def get_thing_shadow(self, thing_name):
        """
        Gets the shadow for an AWS IoT thing.

        :param thing_name: The name of the thing.
        :return: The shadow state as a dictionary.
        """
        import json
        try:
```

```
            response = 
 self.iot_data_client.get_thing_shadow(thingName=thing_name)
            shadow = json.loads(response["payload"].read())
            logger.info("Retrieved shadow for thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot get thing shadow. Resource not found.")
                return None
            logger.error(
                "Couldn't get thing shadow. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return shadow
```

Run an interactive scenario demonstrating IoT basics.

```
class IoTScenario:
    """Runs an interactive scenario that shows how to use AWS IoT."""

    is_interactive = True

    def __init__(self, iot_wrapper, iot_data_client, cfn_client, 
 stack_name="IoTBasicsStack", template_path=None):
        """
        :param iot_wrapper: An instance of the IoTWrapper class.
        :param iot_data_client: A Boto3 IoT Data Plane client.
        :param cfn_client: A Boto3 CloudFormation client.
        :param stack_name: Name for the CloudFormation stack.
        :param template_path: Path to the CloudFormation template file.
        """
        self.iot_wrapper = iot_wrapper
        self.iot_data_client = iot_data_client
        self.cfn_client = cfn_client
        self.thing_name = None
        self.certificate_arn = None
        self.certificate_id = None
        self.rule_name = None
        self.stack_name = stack_name
```

```
        self.template_path = template_path or "../../../scenarios/basics/iot/
iot_usecase/resources/cfn_template.yaml"

    def _deploy_stack(self):
        """Deploy CloudFormation stack and return outputs."""
        with open(self.template_path, "r") as f:
            template_body = f.read()

        try:
            self.cfn_client.create_stack(
                StackName=self.stack_name,
                TemplateBody=template_body,
                Capabilities=["CAPABILITY_NAMED_IAM"]
            )

            waiter = self.cfn_client.get_waiter("stack_create_complete")
            waiter.wait(StackName=self.stack_name)

            response = self.cfn_client.describe_stacks(StackName=self.stack_name)
            outputs = {output["OutputKey"]: output["OutputValue"]
                      for output in response["Stacks"][0]["Outputs"]}
            return outputs["SNSTopicArn"], outputs["RoleArn"]
        except ClientError as err:
            if err.response["Error"]["Code"] == "AlreadyExistsException":
                response =
 self.cfn_client.describe_stacks(StackName=self.stack_name)
                outputs = {output["OutputKey"]: output["OutputValue"]
                          for output in response["Stacks"][0]["Outputs"]}
                return outputs["SNSTopicArn"], outputs["RoleArn"]
            raise

    def _cleanup_stack(self):
        """Delete CloudFormation stack."""
        try:
            self.cfn_client.delete_stack(StackName=self.stack_name)
            waiter = self.cfn_client.get_waiter("stack_delete_complete")
            waiter.wait(StackName=self.stack_name)
            print("CloudFormation stack deleted successfully.")
        except ClientError as err:
            logger.error(f"Failed to delete stack: {err}")

    def run_scenario(self, thing_name, rule_name):
        """
        Runs the IoT basics scenario.
```

```
        :param thing_name: The name of the thing to create.
        :param rule_name: The name of the topic rule to create.
        """
        print("-" * 88)
        print("Welcome to the AWS IoT basics scenario!")
        print("-" * 88)
        print(
            "This scenario demonstrates how to interact with AWS IoT using the
 AWS SDK for Python (Boto3).\n"
            "AWS IoT provides secure, bi-directional communication between
 Internet-connected devices\n"
            "and the AWS cloud. You can manage device connections, process device
 data, and build IoT applications.\n"
        )

        self.thing_name = thing_name
        self.rule_name = rule_name

        try:
            print("\nDeploying CloudFormation stack...")
            sns_topic_arn, role_arn = self._deploy_stack()
            print(f"Stack deployed. SNS Topic: {sns_topic_arn}")

            input("\nNext, we'll create an AWS IoT thing. Press Enter to
 continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("1. Create an AWS IoT thing")
            print("-" * 88)
            response = self.iot_wrapper.create_thing(thing_name)
            print(f"Created thing: {response['thingName']}")
            print(f"Thing ARN: {response['thingArn']}")

            input("\nNext, we'll list things. Press Enter to continue...") if
 self.is_interactive else None
            print("\n" + "-" * 88)
            print("2. List things")
            print("-" * 88)
            things = self.iot_wrapper.list_things()
            print(f"Found {len(things)} thing(s) in your account")
            for thing in things[:5]:  # Show first 5
                print(f"  Thing name: {thing['thingName']}")
```

```
            input("\nNext, we'll generate a device certificate. Press Enter to 
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("3. Generate a device certificate")
            print("-" * 88)
            cert_response = self.iot_wrapper.create_keys_and_certificate()
            self.certificate_arn = cert_response["certificateArn"]
            self.certificate_id = cert_response["certificateId"]
            print(f"Created certificate: {self.certificate_id}")

            input("\nNext, we'll attach the certificate to the thing. Press Enter 
to continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("4. Attach the certificate to the thing")
            print("-" * 88)
            self.iot_wrapper.attach_thing_principal(thing_name, 
self.certificate_arn)
            print(f"Attached certificate to thing: {thing_name}")

            input("\nNext, we'll update the thing shadow. Press Enter to 
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("5. Update the thing shadow")
            print("-" * 88)
            shadow_state = {"state": {"reported": {"temperature": 25, "humidity": 
50}}}
            self.iot_wrapper.update_thing_shadow(thing_name, shadow_state)
            print(f"Updated shadow for thing: {thing_name}")

            input("\nNext, we'll get the thing shadow. Press Enter to 
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("6. Get the thing shadow")
            print("-" * 88)
            shadow = self.iot_wrapper.get_thing_shadow(thing_name)
            print(f"Shadow state: {json.dumps(shadow['state'], indent=2)}")

            input("\nNext, we'll get the AWS IoT endpoint. Press Enter to 
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("7. Get the AWS IoT endpoint")
            print("-" * 88)
            endpoint = self.iot_wrapper.describe_endpoint()
            print(f"IoT endpoint: {endpoint}")
```

```
            input("\nNext, we'll list certificates. Press Enter to continue...")
if self.is_interactive else None
            print("\n" + "-" * 88)
            print("8. List certificates")
            print("-" * 88)
            certificates = self.iot_wrapper.list_certificates()
            print(f"Found {len(certificates)} certificate(s)")
            for cert in certificates:
                print(f"  Certificate ID: {cert['certificateId']}")
                print(f"  Certificate ARN: {cert['certificateArn']}")
                print(f"  Status: {cert['status']}")

            input("\nNext, we'll create a topic rule. Press Enter to
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("9. Create a topic rule")
            print("-" * 88)
            self.iot_wrapper.create_topic_rule(
                rule_name, f"device/{thing_name}/data", sns_topic_arn, role_arn
            )
            print(f"Created topic rule: {rule_name}")

            input("\nNext, we'll list topic rules. Press Enter to continue...")
if self.is_interactive else None
            print("\n" + "-" * 88)
            print("10. List topic rules")
            print("-" * 88)
            rules = self.iot_wrapper.list_topic_rules()
            print(f"Found {len(rules)} topic rule(s)")
            for rule in rules:
                print(f"  Rule name: {rule['ruleName']}")
                print(f"  Rule ARN: {rule['ruleArn']}")

            input("\nNext, we'll configure thing indexing. Press Enter to
continue...") if self.is_interactive else None
            print("\n" + "-" * 88)
            print("11. Configure thing indexing")
            print("-" * 88)
            self.iot_wrapper.update_indexing_configuration()
            print("Enabled thing indexing")
            print("Waiting for indexing to be ready...")
            time.sleep(10)
```

```
            input("\nNext, we'll search for things. Press Enter to continue...")
 if self.is_interactive else None
            print("\n" + "-" * 88)
            print("12. Search for things")
            print("-" * 88)
            try:
                things = self.iot_wrapper.search_index(f"thingName:{thing_name}")
                if things:
                    print(f"Found {len(things)} thing(s) matching the query")
                    for thing in things:
                        print(f"  Thing name: {thing.get('thingName', 'N/A')}")
                        print(f"  Thing ID: {thing.get('thingId', 'N/A')}")
                else:
                    print("No things found. Indexing may take a few minutes.")
            except ClientError as err:
                if err.response["Error"]["Code"] in [
                    "IndexNotReadyException",
                    "InvalidRequestException",
                ]:
                    print("Search index not ready yet. This is expected.")
                else:
                    raise

        except ClientError as err:
            logger.error(
                "Scenario failed: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        finally:
            self._cleanup()

    def _cleanup(self):
        """Cleans up resources created during the scenario."""
        if not self.thing_name:
            return

        print("\n" + "-" * 88)
        print("Cleanup")
        print("-" * 88)

        if q.ask("Do you want to delete the resources? (y/n) ", q.is_yesno):
            try:
```

```
                if self.certificate_arn:
                    print(f"Detaching certificate from thing: {self.thing_name}")
                    self.iot_wrapper.detach_thing_principal(
                        self.thing_name, self.certificate_arn
                    )

                if self.certificate_id:
                    print(f"Deleting certificate: {self.certificate_id}")
                    self.iot_wrapper.delete_certificate(self.certificate_id)

                if self.thing_name:
                    print(f"Deleting thing: {self.thing_name}")
                    self.iot_wrapper.delete_thing(self.thing_name)

                if self.rule_name:
                    print(f"Deleting topic rule: {self.rule_name}")
                    self.iot_wrapper.delete_topic_rule(self.rule_name)

                self._cleanup_stack()
                print("Resources deleted successfully.")
            except ClientError as err:
                logger.error(
                    "Cleanup failed: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
        else:
            print("Resources will remain in your account.")

        print("\n" + "-" * 88)
        print("Thanks for using AWS IoT!")
        print("-" * 88)
```

For a complete list of AWS SDK developer guides and code examples, see [Using AWS IoT with an AWS SDK](). This topic also includes information about getting started and details about previous SDK versions.

Actions for AWS IoT using AWS SDKs

The following code examples demonstrate how to perform individual AWS IoT actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the AWS IoT API Reference.

Examples

Use `AttachThingPrincipal` with an AWS SDK or CLI

The following code examples show how to use `AttachThingPrincipal`.

.NET

SDK for .NET (v4)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Attaches a certificate to an IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
    /// <param name="certificateArn">The ARN of the certificate to attach.</
param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> AttachThingPrincipalAsync(string thingName, string
 certificateArn)
    {
        try
        {
            var request = new AttachThingPrincipalRequest
            {
                ThingName = thingName,
                Principal = certificateArn
            };

            await _amazonIoT.AttachThingPrincipalAsync(request);
            _logger.LogInformation($"Attached certificate {certificateArn} to
 Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot attach certificate - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
```

```
            _logger.LogError($"Couldn't attach certificate to Thing. Here's why:
 {ex.Message}");
            return false;
        }
    }
```

- For API details, see AttachThingPrincipal in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Attach a principal to an AWS IoT thing.
/*!
  \param principal: A principal to attach.
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::attachThingPrincipal(const Aws::String &principal,
                                       const Aws::String &thingName,
                                       const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient client(clientConfiguration);
    Aws::IoT::Model::AttachThingPrincipalRequest request;
    request.SetPrincipal(principal);
    request.SetThingName(thingName);
    Aws::IoT::Model::AttachThingPrincipalOutcome outcome =
 client.AttachThingPrincipal(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully attached principal to thing." << std::endl;
    }
    else {
        std::cerr << "Failed to attach principal to thing." <<
```

```
                    outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see AttachThingPrincipal in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To attach a certificate to your thing

The following `attach-thing-principal` example attaches a certificate to the MyTemperatureSensor thing. The certificate is identified by an ARN. You can find the ARN for a certificate in the AWS IoT console.

```
aws iot attach-thing-principal \
    --thing-name MyTemperatureSensor \
    --principal arn:aws:iot:us-
west-2:123456789012:cert/2e1eb273792174ec2b9bf4e9b37e6c6c692345499506002a35159767055278e8
```

This command produces no output.

For more information, see How to Manage Things with the Registry in the *AWS IoT Developers Guide*.

- For API details, see AttachThingPrincipal in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Attaches a certificate to an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     * @param certificateArn The ARN of the certificate to attach.
     *
     * This method initiates an asynchronous request to attach a certificate to
an IoT Thing.
     * If the request is successful, it prints a confirmation message and
additional information about the Thing.
     * If an exception occurs, it prints the error message.
     */
    public void attachCertificateToThing(String thingName, String certificateArn)
{
        AttachThingPrincipalRequest principalRequest =
AttachThingPrincipalRequest.builder()
            .thingName(thingName)
            .principal(certificateArn)
            .build();

        CompletableFuture<AttachThingPrincipalResponse> future =
getAsyncClient().attachThingPrincipal(principalRequest);
        future.whenComplete((attachResponse, ex) -> {
            if (attachResponse != null &&
attachResponse.sdkHttpResponse().isSuccessful()) {
                System.out.println("Certificate attached to Thing
successfully.");

                // Print additional information about the Thing.
                describeThing(thingName);
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to attach certificate to Thing.
HTTP Status Code: " +
                        attachResponse.sdkHttpResponse().statusCode());
                }
```

```
            }
        });

        future.join();
    }
```

- For API details, see AttachThingPrincipal in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun attachCertificateToThing(
    thingNameVal: String?,
    certificateArn: String?,
) {
    val principalRequest =
        AttachThingPrincipalRequest {
            thingName = thingNameVal
            principal = certificateArn
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.attachThingPrincipal(principalRequest)
        println("Certificate attached to $thingNameVal successfully.")
    }
}
```

- For API details, see AttachThingPrincipal in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def attach_thing_principal(self, thing_name, principal):
        """
        Attaches a certificate to an AWS IoT thing.

        :param thing_name: The name of the thing.
        :param principal: The ARN of the certificate.
        """
        try:
            self.iot_client.attach_thing_principal(
                thingName=thing_name, principal=principal
            )
            logger.info("Attached principal %s to thing %s.", principal,
 thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
```

```
                logger.error("Cannot attach principal. Resource not found.")
                return
            logger.error(
                "Couldn't attach principal to thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see AttachThingPrincipal in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateKeysAndCertificate` with an AWS SDK or CLI

The following code examples show how to use `CreateKeysAndCertificate`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Creates a device certificate for AWS IoT.
    /// </summary>
    /// <returns>The certificate details including ARN and certificate PEM, or
 null if creation failed.</returns>
    public async Task<(string CertificateArn, string CertificatePem, string
 CertificateId)?> CreateKeysAndCertificateAsync()
    {
        try
        {
```

```
            var request = new CreateKeysAndCertificateRequest
            {
                SetAsActive = true
            };

            var response = await
 _amazonIoT.CreateKeysAndCertificateAsync(request);
            _logger.LogInformation($"Created certificate with ARN
 {response.CertificateArn}");
            return (response.CertificateArn, response.CertificatePem,
 response.CertificateId);
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create certificate. Here's why:
 {ex.Message}");
            return null;
        }
    }
```

- For API details, see [CreateKeysAndCertificate](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Create keys and certificate for an Aws IoT device.
//! This routine will save certificates and keys to an output folder, if
 provided.
```

```
/*!
  \param outputFolder: Location for storing output in files, ignored when string
 is empty.
  \param certificateARNResult: A string to receive the ARN of the created
 certificate.
  \param certificateID: A string to receive the ID of the created certificate.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::createKeysAndCertificate(const Aws::String &outputFolder,
                                           Aws::String &certificateARNResult,
                                           Aws::String &certificateID,
                                           const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient client(clientConfiguration);
    Aws::IoT::Model::CreateKeysAndCertificateRequest
 createKeysAndCertificateRequest;

    Aws::IoT::Model::CreateKeysAndCertificateOutcome outcome =
            client.CreateKeysAndCertificate(createKeysAndCertificateRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created a certificate and keys" << std::endl;
        certificateARNResult = outcome.GetResult().GetCertificateArn();
        certificateID = outcome.GetResult().GetCertificateId();
        std::cout << "Certificate ARN: " << certificateARNResult << ",
 certificate ID: "
                  << certificateID << std::endl;

        if (!outputFolder.empty()) {
            std::cout << "Writing certificate and keys to the folder '" <<
 outputFolder
                      << "'." << std::endl;
            std::cout << "Be sure these files are stored securely." << std::endl;

            Aws::String certificateFilePath = outputFolder + "/
certificate.pem.crt";
            std::ofstream certificateFile(certificateFilePath);
            if (!certificateFile.is_open()) {
                std::cerr << "Error opening certificate file, '" <<
 certificateFilePath
                          << "'."
                          << std::endl;
                return false;
            }
```

```
            certificateFile << outcome.GetResult().GetCertificatePem();
            certificateFile.close();

            const Aws::IoT::Model::KeyPair &keyPair =
 outcome.GetResult().GetKeyPair();

            Aws::String privateKeyFilePath = outputFolder + "/private.pem.key";
            std::ofstream privateKeyFile(privateKeyFilePath);
            if (!privateKeyFile.is_open()) {
                std::cerr << "Error opening private key file, '" <<
 privateKeyFilePath
                          << "'."
                          << std::endl;
                return false;
            }
            privateKeyFile << keyPair.GetPrivateKey();
            privateKeyFile.close();

            Aws::String publicKeyFilePath = outputFolder + "/public.pem.key";
            std::ofstream publicKeyFile(publicKeyFilePath);
            if (!publicKeyFile.is_open()) {
                std::cerr << "Error opening public key file, '" <<
 publicKeyFilePath
                          << "'."
                          << std::endl;
                return false;
            }
            publicKeyFile << keyPair.GetPublicKey();
        }
    }
    else {
        std::cerr << "Error creating keys and certificate: "
                  << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see CreateKeysAndCertificate in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To create an RSA key pair and issue an X.509 certificate

The following `create-keys-and-certificate` creates a 2048-bit RSA key pair and issues an X.509 certificate using the issued public key. Because this is the only time that AWS IoT provides the private key for this certificate, be sure to keep it in a secure location.

```
aws iot create-keys-and-certificate \
    --certificate-pem-outfile "myTest.cert.pem" \
    --public-key-outfile "myTest.public.key" \
    --private-key-outfile "myTest.private.key"
```

Output:

```
{
    "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/9894ba17925e663f1d29c23af4582b8e3b7619c31f3fbd93adcb51ae54b83dc2
    "certificateId":
 "9894ba17925e663f1d29c23af4582b8e3b7619c31f3fbd93adcb51ae54b83dc2",
    "certificatePem": "
-----BEGIN CERTIFICATE-----
MIICiTCCEXAMPLE6m7oRw0uXOjANBgkqhkiG9w0BAQUFADCBiDELMAkGA1UEBhMC
VVMxCzAJBgNVBAgEXAMPLEAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6
b24xFDASBgNVBAsTC0lBTSEXAMPLE2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMxHzAd
BgkqhkiG9w0BCQEWEG5vb25lQGFtYEXAMPLEb20wHhcNMTEwNDI1MjA0NTIxWhcN
MTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBhMCEXAMPLEJBgNVBAgTAldBMRAwDgYD
VQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDAEXAMPLEsTC0lBTSBDb25z
b2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEXAMPLE25lQGFt
YXpvbi5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMaK0dn+aEXAMPLE
EXAMPLEfEvySWtC2XADZ4nB+BLYgVIk60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9T
rDHudUZEXAMPLELG5M43q7Wgc/MbQITxOUSQv7c7ugFFDzQGBzZswY6786m86gpE
Ibb3OhjZnzcvQAEXAMPLEWIMm2nrAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4
nUhVVxYUntneD9+h8Mg9qEXAMPLEyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0Fkb
FFBjvSfpJIlJ00zbhNYS5f6GuoEDEXAMPLEBHjJnyp378OD8uTs7fLvjx79LjSTb
NYiytVbZPQUQ5Yaxu2jXnimvw3rrszlaEXAMPLE=
-----END CERTIFICATE-----\n",
    "keyPair": {
        "PublicKey": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\nMMEXAMPLEuuN/
dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\n59VK7cEXAMPLE67GK+y
```

```
+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQ
\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa
\nhJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\nFQIDAQAB
\n-----END PUBLIC KEY-----\n",
        "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\nkey omittted for security
 reasons\n-----END RSA PRIVATE KEY-----\n"
    }
}
```


For more information, see [Create and Register an AWS IoT Device Certificate](#) in the **AWS IoT Developer Guide**.

- For API details, see [CreateKeysAndCertificate](#) in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
    /**
     * Creates an IoT certificate asynchronously.
     *
     * @return The ARN of the created certificate.
     * <p>
     * This method initiates an asynchronous request to create an IoT
 certificate.
     * If the request is successful, it prints the certificate details and
 returns the certificate ARN.
     * If an exception occurs, it prints the error message.
     */
    public String createCertificate() {
        CompletableFuture<CreateKeysAndCertificateResponse> future =
 getAsyncClient().createKeysAndCertificate();
        final String[] certificateArn = {null};
        future.whenComplete((response, ex) -> {
            if (response != null) {
```

```
                    String certificatePem = response.certificatePem();
                    certificateArn[0] = response.certificateArn();

                    // Print the details.
                    System.out.println("\nCertificate:");
                    System.out.println(certificatePem);
                    System.out.println("\nCertificate ARN:");
                    System.out.println(certificateArn[0]);

                } else {
                    Throwable cause = (ex instanceof CompletionException) ?
 ex.getCause() : ex;
                    if (cause instanceof IotException) {
                        System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                    } else {
                        System.err.println("Unexpected error: " +
 cause.getMessage());
                    }
                }
            });

            future.join();
            return certificateArn[0];
        }
```

- For API details, see CreateKeysAndCertificate in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createCertificate(): String? {
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val response = iotClient.createKeysAndCertificate()
```

```
        val certificatePem = response.certificatePem
        val certificateArn = response.certificateArn

        // Print the details.
        println("\nCertificate:")
        println(certificatePem)
        println("\nCertificate ARN:")
        println(certificateArn)
        return certificateArn
    }
}
```

- For API details, see CreateKeysAndCertificate in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)
```

```
    def create_keys_and_certificate(self):
        """
        Creates keys and a certificate for an AWS IoT thing.

        :return: The certificate ID, ARN, and PEM.
        """
        try:
            response =
 self.iot_client.create_keys_and_certificate(setAsActive=True)
            logger.info("Created certificate %s.", response["certificateId"])
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't create keys and certificate. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response
```

- For API details, see CreateKeysAndCertificate in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateThing` with an AWS SDK or CLI

The following code examples show how to use `CreateThing`.

.NET

SDK for .NET (v4)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Creates an AWS IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing to create.</param>
    /// <returns>The ARN of the Thing created, or null if creation failed.</
returns>
    public async Task<string?> CreateThingAsync(string thingName)
    {
        try
        {
            var request = new CreateThingRequest
            {
                ThingName = thingName
            };

            var response = await _amazonIoT.CreateThingAsync(request);
            _logger.LogInformation($"Created Thing {thingName} with ARN
 {response.ThingArn}");
            return response.ThingArn;
        }
        catch (Amazon.IoT.Model.ResourceAlreadyExistsException ex)
        {
            _logger.LogWarning($"Thing {thingName} already exists:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create Thing {thingName}. Here's why:
 {ex.Message}");
            return null;
        }
```

```
    }
```

- For API details, see CreateThing in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Create an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::createThing(const Aws::String &thingName,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::CreateThingRequest createThingRequest;
    createThingRequest.SetThingName(thingName);

    Aws::IoT::Model::CreateThingOutcome outcome = iotClient.CreateThing(
            createThingRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to create thing " << thingName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see CreateThing in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To create a thing record in the registry

The following `create-thing` example creates an entry for a device in the AWS IoT thing registry.

```
aws iot create-thing \
    --thing-name SampleIoTThing
```

Output:

```
{
    "thingName": "SampleIoTThing",
    "thingArn": "arn:aws:iot:us-west-2: 123456789012:thing/SampleIoTThing",
    "thingId": " EXAMPLE1-90ab-cdef-fedc-ba987EXAMPLE "
}
```

Example 2: To define a thing that is associated with a thing type

The following `create-thing` example create a thing that has the specified thing type and its attributes.

```
aws iot create-thing \
    --thing-name "MyLightBulb" \
    --thing-type-name "LightBulb" \
    --attribute-payload "{"attributes": {"wattage":"75", "model":"123"}}"
```

Output:

```
{
    "thingName": "MyLightBulb",
    "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyLightBulb",
    "thingId": "40da2e73-c6af-406e-b415-15acae538797"
}
```

For more information, see How to Manage Things with the Registry and Thing Types in the *AWS IoT Developers Guide*.

- For API details, see CreateThing in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Creates an IoT Thing with the specified name asynchronously.
     *
     * @param thingName The name of the IoT Thing to create.
     *
     * This method initiates an asynchronous request to create an IoT Thing with
 the specified name.
     * If the request is successful, it prints the name of the thing and its ARN
 value.
     * If an exception occurs, it prints the error message.
     */
    public void createIoTThing(String thingName) {
        CreateThingRequest createThingRequest = CreateThingRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<CreateThingResponse> future =
 getAsyncClient().createThing(createThingRequest);
        future.whenComplete((createThingResponse, ex) -> {
            if (createThingResponse != null) {
                System.out.println(thingName + " was successfully created. The
 ARN value is " + createThingResponse.thingArn());
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
```

```
                } else {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                }
            }
        });

        future.join();
    }
```

- For API details, see [CreateThing](#) in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun createIoTThing(thingNameVal: String) {
    val createThingRequest =
        CreateThingRequest {
            thingName = thingNameVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.createThing(createThingRequest)
        println("Created $thingNameVal}")
    }
}
```

- For API details, see [CreateThing](#) in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def create_thing(self, thing_name):
        """
        Creates an AWS IoT thing.

        :param thing_name: The name of the thing to create.
        :return: The name and ARN of the created thing.
        """
        try:
            response = self.iot_client.create_thing(thingName=thing_name)
            logger.info("Created thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceAlreadyExistsException":
                logger.info("Thing %s already exists. Skipping creation.",
 thing_name)
                return None
```

```
            logger.error(
                "Couldn't create thing %s. Here's why: %s: %s",
                thing_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see CreateThing in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateTopicRule` with an AWS SDK or CLI

The following code examples show how to use `CreateTopicRule`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Creates an IoT topic rule.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="snsTopicArn">The ARN of the SNS topic for the action.</
param>
    /// <param name="roleArn">The ARN of the IAM role.</param>
    /// <returns>True if successful, false otherwise.</returns>
```

```
    public async Task<bool> CreateTopicRuleAsync(string ruleName, string
 snsTopicArn, string roleArn)
    {
        try
        {
            var request = new CreateTopicRuleRequest
            {
                RuleName = ruleName,
                TopicRulePayload = new TopicRulePayload
                {
                    Sql = "SELECT * FROM 'topic/subtopic'",
                    Description = $"Rule created by .NET example: {ruleName}",
                    Actions = new List<Amazon.IoT.Model.Action>
                    {
                        new Amazon.IoT.Model.Action
                        {
                            Sns = new SnsAction
                            {
                                TargetArn = snsTopicArn,
                                RoleArn = roleArn
                            }
                        }
                    },
                    RuleDisabled = false
                }
            };

            await _amazonIoT.CreateTopicRuleAsync(request);
            _logger.LogInformation($"Created IoT rule {ruleName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceAlreadyExistsException ex)
        {
            _logger.LogWarning($"Rule {ruleName} already exists: {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't create topic rule. Here's why:
 {ex.Message}");
            return false;
        }
    }
```

- For API details, see CreateTopicRule in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Create an AWS IoT rule with an SNS topic as the target.
/*!
  \param ruleName: The name for the rule.
  \param snsTopic: The SNS topic ARN for the action.
  \param sql: The SQL statement used to query the topic.
  \param roleARN: The IAM role ARN for the action.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool
AwsDoc::IoT::createTopicRule(const Aws::String &ruleName,
                             const Aws::String &snsTopicARN, const Aws::String
 &sql,
                             const Aws::String &roleARN,
                             const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::CreateTopicRuleRequest request;
    request.SetRuleName(ruleName);

    Aws::IoT::Model::SnsAction snsAction;
    snsAction.SetTargetArn(snsTopicARN);
    snsAction.SetRoleArn(roleARN);

    Aws::IoT::Model::Action action;
    action.SetSns(snsAction);
```

```
    Aws::IoT::Model::TopicRulePayload topicRulePayload;
    topicRulePayload.SetSql(sql);
    topicRulePayload.SetActions({action});

    request.SetTopicRulePayload(topicRulePayload);
    auto outcome = iotClient.CreateTopicRule(request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully created topic rule " << ruleName << "." <<
 std::endl;
    }
    else {
        std::cerr << "Error creating topic rule " << ruleName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }
    return outcome.IsSuccess();
}
```

- For API details, see [CreateTopicRule](#) in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To create a rule that sends an Amazon SNS alert

The following `create-topic-rule` example creates a rule that sends an Amazon SNS message when soil moisture level readings, as found in a device shadow, are low.

```
aws iot create-topic-rule \
    --rule-name "LowMoistureRule" \
    --topic-rule-payload file://plant-rule.json
```

The example requires the following JSON code to be saved to a file named `plant-rule.json`:

```
{
    "sql": "SELECT * FROM '$aws/things/MyRPi/shadow/update/accepted' WHERE
 state.reported.moisture = 'low'\n",
    "description": "Sends an alert whenever soil moisture level readings are too
 low.",
    "ruleDisabled": false,
```

```
    "awsIotSqlVersion": "2016-03-23",
    "actions": [{
            "sns": {
                "targetArn": "arn:aws:sns:us-
west-2:123456789012:MyRPiLowMoistureTopic",
                "roleArn": "arn:aws:iam::123456789012:role/service-role/
MyRPiLowMoistureTopicRole",
                "messageFormat": "RAW"
            }
    }]
}
```

This command produces no output.

For more information, see [Creating an AWS IoT Rule](#) in the *AWS IoT Developers Guide*.

- For API details, see [CreateTopicRule](#) in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
    /**
     * Creates an IoT rule asynchronously.
     *
     * @param roleARN The ARN of the IAM role that grants access to the rule's
 actions.
     * @param ruleName The name of the IoT rule.
     * @param action The ARN of the action to perform when the rule is triggered.
     *
     * This method initiates an asynchronous request to create an IoT rule.
     * If the request is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void createIoTRule(String roleARN, String ruleName, String action) {
        String sql = "SELECT * FROM '" + TOPIC + "'";
```

```
        SnsAction action1 = SnsAction.builder()
            .targetArn(action)
            .roleArn(roleARN)
            .build();

        // Create the action.
        Action myAction = Action.builder()
            .sns(action1)
            .build();

        // Create the topic rule payload.
        TopicRulePayload topicRulePayload = TopicRulePayload.builder()
            .sql(sql)
            .actions(myAction)
            .build();

        // Create the topic rule request.
        CreateTopicRuleRequest topicRuleRequest =
 CreateTopicRuleRequest.builder()
            .ruleName(ruleName)
            .topicRulePayload(topicRulePayload)
            .build();

        CompletableFuture<CreateTopicRuleResponse> future =
 getAsyncClient().createTopicRule(topicRuleRequest);
        future.whenComplete((response, ex) -> {
            if (response != null) {
                System.out.println("IoT Rule created successfully.");
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                } else {
                    System.err.println("Failed to create IoT Rule.");
                }
            }
        });

        future.join();
    }
```

- For API details, see CreateTopicRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createIoTRule(
    roleARNVal: String?,
    ruleNameVal: String?,
    action: String?,
) {
    val sqlVal = "SELECT * FROM '$TOPIC '"
    val action1 =
        SnsAction {
            targetArn = action
            roleArn = roleARNVal
        }

    val myAction =
        Action {
            sns = action1
        }

    val topicRulePayloadVal =
        TopicRulePayload {
            sql = sqlVal
            actions = listOf(myAction)
        }

    val topicRuleRequest =
        CreateTopicRuleRequest {
            ruleName = ruleNameVal
            topicRulePayload = topicRulePayloadVal
        }
```

```
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.createTopicRule(topicRuleRequest)
        println("IoT rule created successfully.")
    }
}
```

- For API details, see [CreateTopicRule]() in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository]().

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def create_topic_rule(self, rule_name, topic, sns_action_arn, role_arn):
        """
        Creates an AWS IoT topic rule.

        :param rule_name: The name of the rule.
```

```
        :param topic: The MQTT topic to subscribe to.
        :param sns_action_arn: The ARN of the SNS topic to publish to.
        :param role_arn: The ARN of the IAM role.
        """
        try:
            self.iot_client.create_topic_rule(
                ruleName=rule_name,
                topicRulePayload={
                    "sql": f"SELECT * FROM '{topic}'",
                    "actions": [
                        {"sns": {"targetArn": sns_action_arn, "roleArn":
role_arn}}
                    ],
                },
            )
            logger.info("Created topic rule %s.", rule_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceAlreadyExistsException":
                logger.info("Topic rule %s already exists. Skipping creation.",
rule_name)
                return
            logger.error(
                "Couldn't create topic rule. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see CreateTopicRule in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteCertificate` with an AWS SDK or CLI

The following code examples show how to use `DeleteCertificate`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

```
    /// <summary>
    /// Deletes an IoT certificate.
    /// </summary>
    /// <param name="certificateId">The ID of the certificate to delete.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DeleteCertificateAsync(string certificateId)
    {
        try
        {
            // First, update the certificate to inactive state
            var updateRequest = new UpdateCertificateRequest
            {
                CertificateId = certificateId,
                NewStatus = CertificateStatus.INACTIVE
            };
            await _amazonIoT.UpdateCertificateAsync(updateRequest);

            // Then delete the certificate
            var deleteRequest = new DeleteCertificateRequest
            {
                CertificateId = certificateId
            };

            await _amazonIoT.DeleteCertificateAsync(deleteRequest);
            _logger.LogInformation($"Deleted certificate {certificateId}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot delete certificate - resource not found: 
{ex.Message}");
            return false;
```

```
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't delete certificate. Here's why:
 {ex.Message}");
            return false;
        }
    }
```

- For API details, see DeleteCertificate in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete a certificate.
/*!
  \param certificateID: The ID of a certificate.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::deleteCertificate(const Aws::String &certificateID,
                                    const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::DeleteCertificateRequest request;
    request.SetCertificateId(certificateID);

    Aws::IoT::Model::DeleteCertificateOutcome outcome =
 iotClient.DeleteCertificate(
            request);

    if (outcome.IsSuccess()) {
```

```
        std::cout << "Successfully deleted certificate " << certificateID <<
 std::endl;
    }
    else {
        std::cerr << "Error deleting certificate " << certificateID << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DeleteCertificate in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To delete a device certificate

The following `delete-certificate` example deletes the device certificate with the specified ID.

```
aws iot delete-certificate \
    --certificate-
id c0c57bbc8baaf4631a9a0345c957657f5e710473e3ddbee1428d216d54d53ac9
```

This command produces no output.

For more information, see DeleteCertificate in the *AWS IoT API Reference*.

- For API details, see DeleteCertificate in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Deletes a certificate asynchronously.
     *
     * @param certificateArn The ARN of the certificate to delete.
     *
     * This method initiates an asynchronous request to delete a certificate.
     * If the deletion is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void deleteCertificate(String certificateArn) {
        DeleteCertificateRequest certificateProviderRequest =
 DeleteCertificateRequest.builder()
            .certificateId(extractCertificateId(certificateArn))
            .build();

        CompletableFuture<DeleteCertificateResponse> future =
 getAsyncClient().deleteCertificate(certificateProviderRequest);
        future.whenComplete((voidResult, ex) -> {
            if (ex == null) {
                System.out.println(certificateArn + " was successfully
 deleted.");
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }
```

- For API details, see DeleteCertificate in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteCertificate(certificateArn: String) {
    val certificateProviderRequest =
        DeleteCertificateRequest {
            certificateId = extractCertificateId(certificateArn)
        }
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.deleteCertificate(certificateProviderRequest)
        println("$certificateArn was successfully deleted.")
    }
}
```

- For API details, see DeleteCertificate in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
```

```
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def delete_certificate(self, certificate_id):
        """
        Deletes an AWS IoT certificate.

        :param certificate_id: The ID of the certificate to delete.
        """
        try:
            self.iot_client.update_certificate(
                certificateId=certificate_id, newStatus="INACTIVE"
            )
            self.iot_client.delete_certificate(certificateId=certificate_id)
            logger.info("Deleted certificate %s.", certificate_id)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot delete certificate. Resource not found.")
                return
            logger.error(
                "Couldn't delete certificate. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DeleteCertificate in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use DeleteThing with an AWS SDK or CLI

The following code examples show how to use `DeleteThing`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Deletes an IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing to delete.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> DeleteThingAsync(string thingName)
    {
        try
        {
            var request = new DeleteThingRequest
            {
                ThingName = thingName
            };

            await _amazonIoT.DeleteThingAsync(request);
            _logger.LogInformation($"Deleted Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot delete Thing - resource not found:
 {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't delete Thing. Here's why: {ex.Message}");
            return false;
```

```
        }
    }
```

- For API details, see DeleteThing in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::deleteThing(const Aws::String &thingName,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DeleteThingRequest request;
    request.SetThingName(thingName);
    const auto outcome = iotClient.DeleteThing(request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Error deleting thing " << thingName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DeleteThing in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To display detailed information about a thing

The following `delete-thing` example deletes a thing from the AWS IoT registry for your AWS account.

aws iot delete-thing --thing-name "FourthBulb"

This command produces no output.

For more information, see How to Manage Things with the Registry in the *AWS IoT Developers Guide*.

- For API details, see DeleteThing in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Deletes an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing to delete.
     *
     * This method initiates an asynchronous request to delete an IoT Thing.
     * If the deletion is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
     */
    public void deleteIoTThing(String thingName) {
```

```
        DeleteThingRequest deleteThingRequest = DeleteThingRequest.builder()
            .thingName(thingName)
            .build();

        CompletableFuture<DeleteThingResponse> future =
 getAsyncClient().deleteThing(deleteThingRequest);
        future.whenComplete((voidResult, ex) -> {
            if (ex == null) {
                System.out.println("Deleted Thing " + thingName);
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }
```

- For API details, see DeleteThing in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteIoTThing(thingNameVal: String) {
    val deleteThingRequest =
        DeleteThingRequest {
            thingName = thingNameVal
        }
```

```
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.deleteThing(deleteThingRequest)
        println("Deleted $thingNameVal")
    }
}
```

- For API details, see DeleteThing in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def delete_thing(self, thing_name):
        """
        Deletes an AWS IoT thing.

        :param thing_name: The name of the thing to delete.
        """
```

```
        try:
            self.iot_client.delete_thing(thingName=thing_name)
            logger.info("Deleted thing %s.", thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot delete thing. Resource not found.")
                return
            logger.error(
                "Couldn't delete thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DeleteThing in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteTopicRule` with an AWS SDK or CLI

The following code examples show how to use `DeleteTopicRule`.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete an AWS IoT rule.
/*!
  \param ruleName: The name for the rule.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
```

```
 */
bool AwsDoc::IoT::deleteTopicRule(const Aws::String &ruleName,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DeleteTopicRuleRequest request;
    request.SetRuleName(ruleName);

    Aws::IoT::Model::DeleteTopicRuleOutcome outcome = iotClient.DeleteTopicRule(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted rule " << ruleName << std::endl;
    }
    else {
        std::cerr << "Failed to delete rule " << ruleName <<
                  ": " << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DeleteTopicRule in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To delete a rule

The following `delete-topic-rule` example deletes the specified rule.

```
aws iot delete-topic-rule \
    --rule-name "LowMoistureRule"
```

This command produces no output.

For more information, see Deleting a Rule in the *AWS IoT Developers Guide*.

- For API details, see DeleteTopicRule in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/python/example_code/iot).

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def delete_topic_rule(self, rule_name):
        """
        Deletes an AWS IoT topic rule.

        :param rule_name: The name of the rule to delete.
        """
        try:
            self.iot_client.delete_topic_rule(ruleName=rule_name)
            logger.info("Deleted topic rule %s.", rule_name)
        except ClientError as err:
            logger.error(
                "Couldn't delete topic rule. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
```

```
            raise
```

- For API details, see DeleteTopicRule in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeEndpoint` with an AWS SDK or CLI

The following code examples show how to use `DescribeEndpoint`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Gets the AWS IoT endpoint URL.
    /// </summary>
    /// <returns>The endpoint URL, or null if retrieval failed.</returns>
    public async Task<string?> DescribeEndpointAsync()
    {
        try
        {
            var request = new DescribeEndpointRequest
            {
                EndpointType = "iot:Data-ATS"
            };

            var response = await _amazonIoT.DescribeEndpointAsync(request);
            _logger.LogInformation($"Retrieved endpoint:
{response.EndpointAddress}");
            return response.EndpointAddress;
```

```
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
 {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't describe endpoint. Here's why:
 {ex.Message}");
            return null;
        }
    }
```

- For API details, see [DescribeEndpoint](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Describe the endpoint specific to the AWS account making the call.
/*!
  \param endpointResult: String to receive the endpoint result.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::describeEndpoint(Aws::String &endpointResult,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::String endpoint;
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::DescribeEndpointRequest describeEndpointRequest;
    describeEndpointRequest.SetEndpointType(
            "iot:Data-ATS"); // Recommended endpoint type.
```

```
    Aws::IoT::Model::DescribeEndpointOutcome outcome =
 iotClient.DescribeEndpoint(
            describeEndpointRequest);

    if (outcome.IsSuccess()) {
        std::cout << "Successfully described endpoint." << std::endl;
        endpointResult = outcome.GetResult().GetEndpointAddress();
    }
    else {
        std::cerr << "Error describing endpoint" <<
 outcome.GetError().GetMessage()
                  << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DescribeEndpoint in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To get your current AWS endpoint

The following `describe-endpoint` example retrieves the default AWS endpoint to which all commands are applied.

```
aws iot describe-endpoint
```

Output:

```
{
    "endpointAddress": "abc123defghijk.iot.us-west-2.amazonaws.com"
}
```

For more information, see DescribeEndpoint in the *AWS IoT Developer Guide*.

Example 2: To get your ATS endpoint

The following `describe-endpoint` example retrieves the Amazon Trust Services (ATS) endpoint.

```
aws iot describe-endpoint \
    --endpoint-type iot:Data-ATS
```

Output:

```
{
    "endpointAddress": "abc123defghijk-ats.iot.us-west-2.amazonaws.com"
}
```

For more information, see X.509 Certificates and AWS IoT in the *AWS IoT Developer Guide*.

- For API details, see DescribeEndpoint in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Describes the endpoint of the IoT service asynchronously.
     *
     * @return A CompletableFuture containing the full endpoint URL.
     *
     * This method initiates an asynchronous request to describe the endpoint of
 the IoT service.
     * If the request is successful, it prints and returns the full endpoint URL.
     * If an exception occurs, it prints the error message.
     */
    public String describeEndpoint() {
        CompletableFuture<DescribeEndpointResponse> future =
 getAsyncClient().describeEndpoint(DescribeEndpointRequest.builder().endpointType("iot:Da
ATS").build());
```

```
        final String[] result = {null};

        future.whenComplete((endpointResponse, ex) -> {
            if (endpointResponse != null) {
                String endpointUrl = endpointResponse.endpointAddress();
                String exString = getValue(endpointUrl);
                String fullEndpoint = "https://" + exString + "-ats.iot.us-
east-1.amazonaws.com";

                System.out.println("Full Endpoint URL: " + fullEndpoint);
                result[0] = fullEndpoint;
            } else {
                Throwable cause = (ex instanceof CompletionException) ?
 ex.getCause() : ex;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                }
            }
        });

        future.join();
        return result[0];
    }
```

- For API details, see DescribeEndpoint in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun describeEndpoint(): String? {
```

```
    val request = DescribeEndpointRequest {}
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val endpointResponse = iotClient.describeEndpoint(request)
        val endpointUrl: String? = endpointResponse.endpointAddress
        val exString: String = getValue(endpointUrl)
        val fullEndpoint = "https://$exString-ats.iot.us-east-1.amazonaws.com"
        println("Full endpoint URL: $fullEndpoint")
        return fullEndpoint
    }
}
```

- For API details, see DescribeEndpoint in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def describe_endpoint(self, endpoint_type="iot:Data-ATS"):
```

```
        """
        Gets the AWS IoT endpoint.

        :param endpoint_type: The endpoint type.
        :return: The endpoint.
        """
        try:
            response =
 self.iot_client.describe_endpoint(endpointType=endpoint_type)
            logger.info("Retrieved endpoint %s.", response["endpointAddress"])
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't describe endpoint. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response["endpointAddress"]
```

- For API details, see [DescribeEndpoint](https://docs.aws.amazon.com/goto/boto3/iot-2015-05-28/DescribeEndpoint) in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

```
async fn show_address(client: &Client, endpoint_type: &str) -> Result<(), Error>
 {
    let resp = client
        .describe_endpoint()
        .endpoint_type(endpoint_type)
```

```
        .send()
        .await?;

    println!("Endpoint address: {}", resp.endpoint_address.unwrap());

    println!();

    Ok(())
}
```

- For API details, see DescribeEndpoint in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeThing` with an AWS SDK or CLI

The following code examples show how to use `DescribeThing`.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Describe an AWS IoT thing.
/*!
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::describeThing(const Aws::String &thingName,
                                const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
```

```
    Aws::IoT::Model::DescribeThingRequest request;
    request.SetThingName(thingName);

    Aws::IoT::Model::DescribeThingOutcome outcome =
 iotClient.DescribeThing(request);

    if (outcome.IsSuccess()) {
        const Aws::IoT::Model::DescribeThingResult &result = outcome.GetResult();
        std::cout << "Retrieved thing '" << result.GetThingName() << "'" <<
 std::endl;
        std::cout << "thingArn: " << result.GetThingArn() << std::endl;
        std::cout << result.GetAttributes().size() << " attribute(s) retrieved"
                  << std::endl;
        for (const auto &attribute: result.GetAttributes()) {
            std::cout << "  attribute: " << attribute.first << "=" <<
 attribute.second
                      << std::endl;
        }
    }
    else {
        std::cerr << "Error describing thing " << thingName << ": " <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DescribeThing in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To display detailed information about a thing

The following describe-thing example display information about a thing (device) that is defined in the AWS IoT registry for your AWS account.

aws iot describe-thing --thing-name "MyLightBulb"

Output:

```
{
    "defaultClientId": "MyLightBulb",
    "thingName": "MyLightBulb",
    "thingId": "40da2e73-c6af-406e-b415-15acae538797",
    "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyLightBulb",
    "thingTypeName": "LightBulb",
    "attributes": {
        "model": "123",
        "wattage": "75"
    },
    "version": 1
}
```


For more information, see How to Manage Things with the Registry in the *AWS IoT Developers Guide*.

- For API details, see DescribeThing in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Describes an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     *
     * This method initiates an asynchronous request to describe an IoT Thing.
     * If the request is successful, it prints the Thing details.
     * If an exception occurs, it prints the error message.
     */
    private void describeThing(String thingName) {
        DescribeThingRequest thingRequest = DescribeThingRequest.builder()
            .thingName(thingName)
            .build();
```

```
        CompletableFuture<DescribeThingResponse> future =
getAsyncClient().describeThing(thingRequest);
        future.whenComplete((describeResponse, ex) -> {
            if (describeResponse != null) {
                System.out.println("Thing Details:");
                System.out.println("Thing Name: " +
describeResponse.thingName());
                System.out.println("Thing ARN: " + describeResponse.thingArn());
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
cause.getMessage());
                } else {
                    System.err.println("Failed to describe Thing.");
                }
            }
        });

        future.join();
    }
```

- For API details, see DescribeThing in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun describeThing(thingNameVal: String) {
    val thingRequest =
        DescribeThingRequest {
```

```
            thingName = thingNameVal
        }

    // Print Thing details.
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val describeResponse = iotClient.describeThing(thingRequest)
        println("Thing details:")
        println("Thing name: ${describeResponse.thingName}")
        println("Thing ARN:  ${describeResponse.thingArn}")
    }
}
```

- For API details, see DescribeThing in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DetachThingPrincipal` with an AWS SDK or CLI

The following code examples show how to use `DetachThingPrincipal`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Detaches a certificate from an IoT Thing.
    /// </summary>
    /// <param name="thingName">The name of the Thing.</param>
    /// <param name="certificateArn">The ARN of the certificate to detach.</
param>
    /// <returns>True if successful, false otherwise.</returns>
```

```
    public async Task<bool> DetachThingPrincipalAsync(string thingName, string
certificateArn)
    {
        try
        {
            var request = new DetachThingPrincipalRequest
            {
                ThingName = thingName,
                Principal = certificateArn
            };

            await _amazonIoT.DetachThingPrincipalAsync(request);
            _logger.LogInformation($"Detached certificate {certificateArn} from
Thing {thingName}");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot detach certificate - resource not found:
{ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't detach certificate from Thing. Here's
why: {ex.Message}");
            return false;
        }
    }
```

- For API details, see [DetachThingPrincipal](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
//! Detach a principal from an AWS IoT thing.
/*!
  \param principal: A principal to detach.
  \param thingName: The name for the thing.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::detachThingPrincipal(const Aws::String &principal,
                                       const Aws::String &thingName,
                                       const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::DetachThingPrincipalRequest detachThingPrincipalRequest;
    detachThingPrincipalRequest.SetThingName(thingName);
    detachThingPrincipalRequest.SetPrincipal(principal);

    Aws::IoT::Model::DetachThingPrincipalOutcome outcome =
 iotClient.DetachThingPrincipal(
            detachThingPrincipalRequest);

    if (outcome.IsSuccess()) {
        std::cout << "Successfully detached principal " << principal << " from
 thing "
                  << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to detach principal " << principal << " from thing "
                  << thingName << ": "
                  << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see DetachThingPrincipal in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To detach a certificate/principal from a thing

The following `detach-thing-principal` example removes a certificate that represents a principal from the specified thing.

```
aws iot detach-thing-principal \
    --thing-name "MyLightBulb" \
    --principal "arn:aws:iot:us-
west-2:123456789012:cert/604c48437a57b7d5fc5d137c5be75011c6ee67c9a6943683a1acb4b1626bac36
```

This command produces no output.

For more information, see How to Manage Things with the Registry in the *AWS IoT Developers Guide*.

- For API details, see DetachThingPrincipal in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Detaches a principal (certificate) from an IoT Thing asynchronously.
     *
     * @param thingName The name of the IoT Thing.
     * @param certificateArn The ARN of the certificate to detach.
     *
     * This method initiates an asynchronous request to detach a certificate from
 an IoT Thing.
     * If the detachment is successful, it prints a confirmation message.
     * If an exception occurs, it prints the error message.
```

```
     */
    public void detachThingPrincipal(String thingName, String certificateArn) {
        DetachThingPrincipalRequest thingPrincipalRequest =
DetachThingPrincipalRequest.builder()
            .principal(certificateArn)
            .thingName(thingName)
            .build();

        CompletableFuture<DetachThingPrincipalResponse> future =
getAsyncClient().detachThingPrincipal(thingPrincipalRequest);
        future.whenComplete((voidResult, ex) -> {
            if (ex == null) {
                System.out.println(certificateArn + " was successfully removed
from " + thingName);
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
cause).awsErrorDetails().errorMessage());
                } else {
                    System.err.println("Unexpected error: " + ex.getMessage());
                }
            }
        });

        future.join();
    }
```

- For API details, see DetachThingPrincipal in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun detachThingPrincipal(
```

```
    thingNameVal: String,
    certificateArn: String,
) {
    val thingPrincipalRequest =
        DetachThingPrincipalRequest {
            principal = certificateArn
            thingName = thingNameVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        iotClient.detachThingPrincipal(thingPrincipalRequest)
        println("$certificateArn was successfully removed from $thingNameVal")
    }
}
```

- For API details, see DetachThingPrincipal in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
```

```
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def detach_thing_principal(self, thing_name, principal):
        """
        Detaches a certificate from an AWS IoT thing.

        :param thing_name: The name of the thing.
        :param principal: The ARN of the certificate.
        """
        try:
            self.iot_client.detach_thing_principal(
                thingName=thing_name, principal=principal
            )
            logger.info("Detached principal %s from thing %s.", principal, 
thing_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error("Cannot detach principal. Resource not found.")
                return
            logger.error(
                "Couldn't detach principal from thing. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DetachThingPrincipal in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListCertificates` with an AWS SDK or CLI

The following code examples show how to use `ListCertificates`.

.NET

SDK for .NET (v4)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Lists all certificates associated with the account.
    /// </summary>
    /// <returns>List of certificate information, or empty list if listing
 failed.</returns>
    public async Task<List<Certificate>> ListCertificatesAsync()
    {
        try
        {
            var request = new ListCertificatesRequest();
            var response = await _amazonIoT.ListCertificatesAsync(request);

            _logger.LogInformation($"Retrieved {response.Certificates.Count}
 certificates");
            return response.Certificates;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
 {ex.Message}");
            return new List<Certificate>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't list certificates. Here's why:
 {ex.Message}");
            return new List<Certificate>();
        }
    }
```

- For API details, see ListCertificates in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/cpp/example_code/iot#code-examples).

```
//! List certificates registered in the AWS account making the call.
/*!
   \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::listCertificates(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::ListCertificatesRequest request;

    Aws::Vector<Aws::IoT::Model::Certificate> allCertificates;
    Aws::String marker; // Used to paginate results.
    do {
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::IoT::Model::ListCertificatesOutcome outcome =
 iotClient.ListCertificates(
                request);

        if (outcome.IsSuccess()) {
            const Aws::IoT::Model::ListCertificatesResult &result =
 outcome.GetResult();
            marker = result.GetNextMarker();
            allCertificates.insert(allCertificates.end(),
                                   result.GetCertificates().begin(),
                                   result.GetCertificates().end());
        }
        else {
            std::cerr << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
```

```
            return false;
        }
    } while (!marker.empty());

    std::cout << allCertificates.size() << " certificate(s) found." << std::endl;

    for (auto &certificate: allCertificates) {
        std::cout << "Certificate ID: " << certificate.GetCertificateId() <<
 std::endl;
        std::cout << "Certificate ARN: " << certificate.GetCertificateArn()
                  << std::endl;
        std::cout << std::endl;
    }

    return true;
}
```

- For API details, see ListCertificates in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To list the certificates registered in your AWS account

The following `list-certificates` example lists all certificates registered in your account. If you have more than the default paging limit of 25, you can use the `nextMarker` response value from this command and supply it to the next command to get the next batch of results. Repeat until `nextMarker` returns without a value.

```
aws iot list-certificates
```

Output:

```
{
    "certificates": [
        {
            "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/604c48437a57b7d5fc5d137c5be75011c6ee67c9a6943683a1acb4b1626bac36
            "certificateId":
 "604c48437a57b7d5fc5d137c5be75011c6ee67c9a6943683a1acb4b1626bac36",
```

```
            "status": "ACTIVE",
            "creationDate": 1556810537.617
        },
        {
            "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/262a1ac8a7d8aa72f6e96e365480f7313aa9db74b8339ec65d34dc3074e1c31e
            "certificateId":
 "262a1ac8a7d8aa72f6e96e365480f7313aa9db74b8339ec65d34dc3074e1c31e",
            "status": "ACTIVE",
            "creationDate": 1546447050.885
        },
        {
            "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
b193ab7162c0fadca83246d24fa090300a1236fe58137e121b011804d8ac1d6b",
            "certificateId":
 "b193ab7162c0fadca83246d24fa090300a1236fe58137e121b011804d8ac1d6b",
            "status": "ACTIVE",
            "creationDate": 1546292258.322
        },
        {
            "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/7aebeea3845d14a44ec80b06b8b78a89f3f8a706974b8b34d18f5adf0741db42
            "certificateId":
 "7aebeea3845d14a44ec80b06b8b78a89f3f8a706974b8b34d18f5adf0741db42",
            "status": "ACTIVE",
            "creationDate": 1541457693.453
        },
        {
            "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/54458aa39ebb3eb39c91ffbbdcc3a6ca1c7c094d1644b889f735a6fc2cd9a7e3
            "certificateId":
 "54458aa39ebb3eb39c91ffbbdcc3a6ca1c7c094d1644b889f735a6fc2cd9a7e3",
            "status": "ACTIVE",
            "creationDate": 1541113568.611
        },
        {
            "certificateArn": "arn:aws:iot:us-
west-2:123456789012:cert/4f0ba725787aa94d67d2fca420eca022242532e8b3c58e7465c7778b443fd65e
            "certificateId":
 "4f0ba725787aa94d67d2fca420eca022242532e8b3c58e7465c7778b443fd65e",
            "status": "ACTIVE",
            "creationDate": 1541022751.983
        }
    ]
```

```
}
```

- For API details, see ListCertificates in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Lists all certificates asynchronously.
     *
     * This method initiates an asynchronous request to list all certificates.
     * If the request is successful, it prints the certificate IDs and ARNs.
     * If an exception occurs, it prints the error message.
     */
    public void listCertificates() {
        CompletableFuture<ListCertificatesResponse> future =
 getAsyncClient().listCertificates();
        future.whenComplete((response, ex) -> {
            if (response != null) {
                List<Certificate> certList = response.certificates();
                for (Certificate cert : certList) {
                    System.out.println("Cert id: " + cert.certificateId());
                    System.out.println("Cert Arn: " + cert.certificateArn());
                }
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                } else {
                    System.err.println("Failed to list certificates.");
                }
```

```
            }
        });

        future.join();
    }
```

- For API details, see ListCertificates in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun listCertificates() {
    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val response = iotClient.listCertificates()
        val certList = response.certificates
        certList?.forEach { cert ->
            println("Cert id: ${cert.certificateId}")
            println("Cert Arn: ${cert.certificateArn}")
        }
    }
}
```

- For API details, see ListCertificates in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def list_certificates(self):
        """
        Lists AWS IoT certificates.

        :return: The list of certificates.
        """
        try:
            certificates = []
            paginator = self.iot_client.get_paginator("list_certificates")
            for page in paginator.paginate():
                certificates.extend(page["certificates"])
            logger.info("Retrieved %s certificates.", len(certificates))
            return certificates
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
```

```
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't list certificates. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
```

- For API details, see ListCertificates in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListThings` with an AWS SDK or CLI

The following code examples show how to use `ListThings`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Lists IoT Things with pagination support.
    /// </summary>
    /// <returns>List of Things, or empty list if listing failed.</returns>
    public async Task<List<ThingAttribute>> ListThingsAsync()
    {
        try
        {
            // Use pages of 10.
            var request = new ListThingsRequest()
```

```
            {
                MaxResults = 10
            };
            var response = await _amazonIoT.ListThingsAsync(request);

            // Since there is not a built-in paginator, use the NextMarker to
paginate.
            bool hasMoreResults = true;

            var things = new List<ThingAttribute>();
            while (hasMoreResults)
            {
                things.AddRange(response.Things);

                // If NextMarker is not null, there are more results. Get the
next page of results.
                if (!String.IsNullOrEmpty(response.NextMarker))
                {
                    request.Marker = response.NextMarker;
                    response = await _amazonIoT.ListThingsAsync(request);
                }
                else
                    hasMoreResults = false;
            }

            _logger.LogInformation($"Retrieved {things.Count} Things");
            return things;
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later:
{ex.Message}");
            return new List<ThingAttribute>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't list Things. Here's why: {ex.Message}");
            return new List<ThingAttribute>();
        }
    }
```

- For API details, see ListThings in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

Example 1: To list all things in the registry

The following `list-things` example lists the things (devices) that are defined in the AWS IoT registry for your AWS account.

```
aws iot list-things
```

Output:


```
{
    "things": [
        {
            "thingName": "ThirdBulb",
            "thingTypeName": "LightBulb",
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/ThirdBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 2
        },
        {
            "thingName": "MyOtherLightBulb",
            "thingTypeName": "LightBulb",
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/
MyOtherLightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 3
        },
        {
            "thingName": "MyLightBulb",
            "thingTypeName": "LightBulb",
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyLightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
```

```
            },
            "version": 1
        },
        {
        "thingName": "SampleIoTThing",
        "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/SampleIoTThing",
        "attributes": {},
        "version": 1
        }
    ]
}
```

Example 2: To list the defined things that have a specific attribute

The following `list-things` example displays a list of things that have an attribute named `wattage`.

```
aws iot list-things \
    --attribute-name wattage
```

Output:

```
{
    "things": [
        {
            "thingName": "MyLightBulb",
            "thingTypeName": "LightBulb",
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyLightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "version": 1
        },
        {
            "thingName": "MyOtherLightBulb",
            "thingTypeName": "LightBulb",
            "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/
MyOtherLightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
```

```
            "version": 3
        }
    ]
}
```

For more information, see [How to Manage Things with the Registry](#) in the *AWS IoT Developers Guide*.

- For API details, see [ListThings](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def list_things(self):
        """
        Lists AWS IoT things.

        :return: The list of things.
```

```
        """
        try:
            things = []
            paginator = self.iot_client.get_paginator("list_things")
            for page in paginator.paginate():
                things.extend(page["things"])
            logger.info("Retrieved %s things.", len(things))
            return things
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't list things. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
```

- For API details, see ListThings in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn show_things(client: &Client) -> Result<(), Error> {
    let resp = client.list_things().send().await?;

    println!("Things:");

    for thing in resp.things.unwrap() {
        println!(
            "  Name:  {}",
```

```
                thing.thing_name.as_deref().unwrap_or_default()
            );
            println!(
                "   Type:   {}",
                thing.thing_type_name.as_deref().unwrap_or_default()
            );
            println!(
                "   ARN:    {}",
                thing.thing_arn.as_deref().unwrap_or_default()
            );
            println!();
        }

        println!();

        Ok(())
    }
```

- For API details, see ListThings in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `SearchIndex` with an AWS SDK or CLI

The following code examples show how to use `SearchIndex`.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Searches for IoT Things using the search index.
```

```
    /// </summary>
    /// <param name="queryString">The search query string.</param>
    /// <returns>List of Things that match the search criteria, or empty list if 
search failed.</returns>
    public async Task<List<ThingDocument>> SearchIndexAsync(string queryString)
    {
        try
        {
            // First, try to perform the search
            var request = new SearchIndexRequest
            {
                QueryString = queryString
            };

            var response = await _amazonIoT.SearchIndexAsync(request);
            _logger.LogInformation($"Search found {response.Things.Count} 
Things");
            return response.Things;
        }
        catch (Amazon.IoT.Model.IndexNotReadyException ex)
        {
            _logger.LogWarning($"Search index not ready, setting up indexing 
configuration: {ex.Message}");
            return await SetupIndexAndRetrySearchAsync(queryString);
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex) when 
(ex.Message.Contains("index") || ex.Message.Contains("Index"))
        {
            _logger.LogWarning($"Search index not configured, setting up indexing 
configuration: {ex.Message}");
            return await SetupIndexAndRetrySearchAsync(queryString);
        }
        catch (Amazon.IoT.Model.ThrottlingException ex)
        {
            _logger.LogWarning($"Request throttled, please try again later: 
{ex.Message}");
            return new List<ThingDocument>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't search index. Here's why: {ex.Message}");
            return new List<ThingDocument>();
        }
    }
```

```
    /// <summary>
    /// Sets up the indexing configuration and retries the search after waiting
 for the index to be ready.
    /// </summary>
    /// <param name="queryString">The search query string.</param>
    /// <returns>List of Things that match the search criteria, or empty list if
 setup/search failed.</returns>
    private async Task<List<ThingDocument>> SetupIndexAndRetrySearchAsync(string
 queryString)
    {
        try
        {
            // Update indexing configuration to REGISTRY mode
            _logger.LogInformation("Setting up IoT search indexing
 configuration...");
            await _amazonIoT.UpdateIndexingConfigurationAsync(
                new UpdateIndexingConfigurationRequest()
                {
                    ThingIndexingConfiguration = new ThingIndexingConfiguration()
                    {
                        ThingIndexingMode = ThingIndexingMode.REGISTRY
                    }
                });

            _logger.LogInformation("Indexing configuration updated. Waiting for
 index to be ready...");

            // Wait for the index to be set up - this can take some time
            const int maxRetries = 10;
            const int retryDelaySeconds = 10;

            for (int attempt = 1; attempt <= maxRetries; attempt++)
            {
                try
                {
                    _logger.LogInformation($"Waiting for index to be ready
 (attempt {attempt}/{maxRetries})...");
                    await Task.Delay(TimeSpan.FromSeconds(retryDelaySeconds));

                    // Try to get the current indexing configuration to see if
 it's ready
                    var configResponse = await
 _amazonIoT.GetIndexingConfigurationAsync(new GetIndexingConfigurationRequest());
```

```
                    if
(configResponse.ThingIndexingConfiguration?.ThingIndexingMode ==
ThingIndexingMode.REGISTRY)
                    {
                        // Try the search again
                        var request = new SearchIndexRequest
                        {
                            QueryString = queryString
                        };

                        var response = await
_amazonIoT.SearchIndexAsync(request);
                        _logger.LogInformation($"Search found
{response.Things.Count} Things after index setup");
                        return response.Things;
                    }
                }
                catch (Amazon.IoT.Model.IndexNotReadyException)
                {
                    // Index still not ready, continue waiting
                    _logger.LogInformation("Index still not ready, continuing to
wait...");
                    continue;
                }
                catch (Amazon.IoT.Model.InvalidRequestException ex) when
(ex.Message.Contains("index") || ex.Message.Contains("Index"))
                {
                    // Index still not ready, continue waiting
                    _logger.LogInformation("Index still not ready, continuing to
wait...");
                    continue;
                }
            }

            _logger.LogWarning("Timeout waiting for search index to be ready
after configuration update");
            return new List<ThingDocument>();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Couldn't set up search index configuration. Here's
why: {ex.Message}");
            return new List<ThingDocument>();
        }
```

```
    }
```

- For API details, see SearchIndex in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Query the AWS IoT fleet index.
//! For query information, see https://docs.aws.amazon.com/iot/latest/
developerguide/query-syntax.html
/*!
  \param: query: The query string.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::searchIndex(const Aws::String &query,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::SearchIndexRequest request;
    request.SetQueryString(query);

    Aws::Vector<Aws::IoT::Model::ThingDocument> allThingDocuments;
    Aws::String nextToken; // Used for pagination.
    do {
        if (!nextToken.empty()) {
            request.SetNextToken(nextToken);
        }

        Aws::IoT::Model::SearchIndexOutcome outcome =
 iotClient.SearchIndex(request);

        if (outcome.IsSuccess()) {
```

```
            const Aws::IoT::Model::SearchIndexResult &result =
 outcome.GetResult();
            allThingDocuments.insert(allThingDocuments.end(),
                                     result.GetThings().cbegin(),
                                     result.GetThings().cend());
            nextToken = result.GetNextToken();

        }
        else {
            std::cerr << "Error in SearchIndex: " <<
 outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!nextToken.empty());

    std::cout << allThingDocuments.size() << " thing document(s) found." <<
 std::endl;
    for (const auto thingDocument: allThingDocuments) {
        std::cout << "  Thing name: " << thingDocument.GetThingName() << "."
                  << std::endl;
    }
    return true;
}
```

- For API details, see SearchIndex in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To query the thing index

The following `search-index` example queries the `AWS_Things` index for things that have a type of `LightBulb`.

```
aws iot search-index \
    --index-name "AWS_Things" \
    --query-string "thingTypeName:LightBulb"
```

Output:

```
{
    "things": [
        {
            "thingName": "MyLightBulb",
            "thingId": "40da2e73-c6af-406e-b415-15acae538797",
            "thingTypeName": "LightBulb",
            "thingGroupNames": [
                "LightBulbs",
                "DeadBulbs"
            ],
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "connectivity": {
                "connected": false
            }
        },
        {
            "thingName": "ThirdBulb",
            "thingId": "615c8455-33d5-40e8-95fd-3ee8b24490af",
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "connectivity": {
                "connected": false
            }
        },
        {
            "thingName": "MyOtherLightBulb",
            "thingId": "6dae0d3f-40c1-476a-80c4-1ed24ba6aa11",
            "thingTypeName": "LightBulb",
            "attributes": {
                "model": "123",
                "wattage": "75"
            },
            "connectivity": {
                "connected": false
            }
        }
    ]
```

```
}
```

For more information, see [Managing Thing Indexing](#) in the *AWS IoT Developer Guide*.

- For API details, see [SearchIndex](#) in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
    /**
     * Searches for IoT Things asynchronously based on a query string.
     *
     * @param queryString The query string to search for Things.
     *
     * This method initiates an asynchronous request to search for IoT Things.
     * If the request is successful and Things are found, it prints their IDs.
     * If no Things are found, it prints a message indicating so.
     * If an exception occurs, it prints the error message.
     */
    public void searchThings(String queryString) {
        SearchIndexRequest searchIndexRequest = SearchIndexRequest.builder()
            .queryString(queryString)
            .build();

        CompletableFuture<SearchIndexResponse> future =
 getAsyncClient().searchIndex(searchIndexRequest);
        future.whenComplete((searchIndexResponse, ex) -> {
            if (searchIndexResponse != null) {
                // Process the result.
                if (searchIndexResponse.things().isEmpty()) {
                    System.out.println("No things found.");
                } else {
                    searchIndexResponse.things().forEach(thing ->
 System.out.println("Thing id found using search is " + thing.thingId()));
                }
```

```
            } else {
                Throwable cause = ex != null ? ex.getCause() : null;
                if (cause instanceof IotException) {
                    System.err.println(((IotException)
 cause).awsErrorDetails().errorMessage());
                } else if (cause != null) {
                    System.err.println("Unexpected error: " +
 cause.getMessage());
                } else {
                    System.err.println("Failed to search for IoT Things.");
                }
            }
        });

        future.join();
    }
```

- For API details, see [SearchIndex](#) in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun searchThings(queryStringVal: String?) {
    val searchIndexRequest =
        SearchIndexRequest {
            queryString = queryStringVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        val searchIndexResponse = iotClient.searchIndex(searchIndexRequest)
        if (searchIndexResponse.things?.isEmpty() == true) {
            println("No things found.")
        } else {
            searchIndexResponse.things
```

```
                ?.forEach { thing -> println("Thing id found using search is
 ${thing.thingId}") }
        }
    }
}
```

- For API details, see SearchIndex in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client

    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def search_index(self, query):
        """
        Searches the AWS IoT index.

        :param query: The search query.
        :return: The list of things found.
```

```
        """
        try:
            response = self.iot_client.search_index(queryString=query)
            logger.info("Found %s things.", len(response.get("things", [])))
        except ClientError as err:
            if err.response["Error"]["Code"] == "ThrottlingException":
                logger.error("Request throttled. Please try again later.")
            else:
                logger.error(
                    "Couldn't search index. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response.get("things", [])
```

- For API details, see SearchIndex in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateIndexingConfiguration` with an AWS SDK or CLI

The following code examples show how to use `UpdateIndexingConfiguration`.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Update the indexing configuration.
/*!
```

```
  \param thingIndexingConfiguration: A ThingIndexingConfiguration object which is
 ignored if not set.
  \param thingGroupIndexingConfiguration: A ThingGroupIndexingConfiguration
 object which is ignored if not set.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::updateIndexingConfiguration(
        const Aws::IoT::Model::ThingIndexingConfiguration
 &thingIndexingConfiguration,
        const Aws::IoT::Model::ThingGroupIndexingConfiguration
 &thingGroupIndexingConfiguration,
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);

    Aws::IoT::Model::UpdateIndexingConfigurationRequest request;

    if (thingIndexingConfiguration.ThingIndexingModeHasBeenSet()) {
        request.SetThingIndexingConfiguration(thingIndexingConfiguration);
    }

    if (thingGroupIndexingConfiguration.ThingGroupIndexingModeHasBeenSet()) {

 request.SetThingGroupIndexingConfiguration(thingGroupIndexingConfiguration);
    }

    Aws::IoT::Model::UpdateIndexingConfigurationOutcome outcome =
 iotClient.UpdateIndexingConfiguration(
            request);

    if (outcome.IsSuccess()) {
        std::cout << "UpdateIndexingConfiguration succeeded." << std::endl;
    }
    else {
        std::cerr << "UpdateIndexingConfiguration failed."
                  << outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see UpdateIndexingConfiguration in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To enable thing indexing

The following `update-indexing-configuration` example enables thing indexing to support searching registry data, shadow data, and thing connectivity status using the AWS_Things index.

```
aws iot update-indexing-configuration
    --thing-indexing-
configuration thingIndexingMode=REGISTRY_AND_SHADOW,thingConnectivityIndexingMode=STATUS
```

This command produces no output.

For more information, see Managing Thing Indexing in the *AWS IoT Developers Guide*.

- For API details, see UpdateIndexingConfiguration in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class IoTWrapper:
    """Encapsulates AWS IoT actions."""

    def __init__(self, iot_client, iot_data_client=None):
        """
        :param iot_client: A Boto3 AWS IoT client.
        :param iot_data_client: A Boto3 AWS IoT Data Plane client.
        """
        self.iot_client = iot_client
        self.iot_data_client = iot_data_client
```

```
    @classmethod
    def from_client(cls):
        iot_client = boto3.client("iot")
        iot_data_client = boto3.client("iot-data")
        return cls(iot_client, iot_data_client)

    def update_indexing_configuration(self):
        """
        Updates the AWS IoT indexing configuration to enable thing indexing.
        """
        try:
            self.iot_client.update_indexing_configuration(
                thingIndexingConfiguration={"thingIndexingMode": "REGISTRY"}
            )
            logger.info("Updated indexing configuration.")
        except ClientError as err:
            logger.error(
                "Couldn't update indexing configuration. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see UpdateIndexingConfiguration in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateThing` with an AWS SDK or CLI

The following code examples show how to use `UpdateThing`.

.NET

SDK for .NET (v4)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Updates an IoT Thing with attributes.
    /// </summary>
    /// <param name="thingName">The name of the Thing to update.</param>
    /// <param name="attributes">Dictionary of attributes to add.</param>
    /// <returns>True if successful, false otherwise.</returns>
    public async Task<bool> UpdateThingAsync(string thingName, Dictionary<string, string> attributes)
    {
        try
        {
            var request = new UpdateThingRequest
            {
                ThingName = thingName,
                AttributePayload = new AttributePayload
                {
                    Attributes = attributes,
                    Merge = true
                }
            };

            await _amazonIoT.UpdateThingAsync(request);
            _logger.LogInformation($"Updated Thing {thingName} with attributes");
            return true;
        }
        catch (Amazon.IoT.Model.ResourceNotFoundException ex)
        {
            _logger.LogError($"Cannot update Thing - resource not found: {ex.Message}");
            return false;
        }
        catch (Exception ex)
```

```
        {
            _logger.LogError($"Couldn't update Thing attributes. Here's why:
 {ex.Message}");
            return false;
        }
    }
```

- For API details, see UpdateThing in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Update an AWS IoT thing with attributes.
/*!
  \param thingName: The name for the thing.
  \param attributeMap: A map of key/value attributes/
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::IoT::updateThing(const Aws::String &thingName,
                              const std::map<Aws::String, Aws::String>
 &attributeMap,
                              const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::IoT::IoTClient iotClient(clientConfiguration);
    Aws::IoT::Model::UpdateThingRequest request;
    request.SetThingName(thingName);
    Aws::IoT::Model::AttributePayload attributePayload;
    for (const auto &attribute: attributeMap) {
        attributePayload.AddAttributes(attribute.first, attribute.second);
    }
    request.SetAttributePayload(attributePayload);

    Aws::IoT::Model::UpdateThingOutcome outcome = iotClient.UpdateThing(request);
```

```
    if (outcome.IsSuccess()) {
        std::cout << "Successfully updated thing " << thingName << std::endl;
    }
    else {
        std::cerr << "Failed to update thing " << thingName << ":" <<
                  outcome.GetError().GetMessage() << std::endl;
    }

    return outcome.IsSuccess();
}
```

- For API details, see UpdateThing in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To associate a thing with a thing type

The following `update-thing` example associates a thing in the AWS IoT registry with a thing type. When you make the association, you provide values for the attributes defined by the thing type.

```
aws iot update-thing \
    --thing-name "MyOtherLightBulb" \
    --thing-type-name "LightBulb" \
    --attribute-payload "{"attributes": {"wattage":"75", "model":"123"}}"
```

This command does not produce output. Use the `describe-thing` command to see the result.

For more information, see Thing Types in the *AWS IoT Developers Guide*.

- For API details, see UpdateThing in *AWS CLI Command Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun updateThing(thingNameVal: String?) {
    val newLocation = "Office"
    val newFirmwareVersion = "v2.0"
    val attMap: MutableMap<String, String> = HashMap()
    attMap["location"] = newLocation
    attMap["firmwareVersion"] = newFirmwareVersion

    val attributePayloadVal =
        AttributePayload {
            attributes = attMap
        }

    val updateThingRequest =
        UpdateThingRequest {
            thingName = thingNameVal
            attributePayload = attributePayloadVal
        }

    IotClient.fromEnvironment { region = "us-east-1" }.use { iotClient ->
        // Update the IoT thing attributes.
        iotClient.updateThing(updateThingRequest)
        println("$thingNameVal attributes updated successfully.")
    }
}
```

- For API details, see UpdateThing in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using AWS IoT with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

AWS IoT quotas

You can find information about AWS IoT quotas in the *AWS General Reference*.

- For AWS IoT Core quotas information, see AWS IoT Core Endpoints and Quotas.
- For AWS IoT Device Management quotas information, see AWS IoT Device Management Endpoints and Quotas.
- For AWS IoT Device Defender quotas information, see AWS IoT Device Defender Endpoints and Quotas.

AWS IoT Core pricing

You can find information about AWS IoT Core pricing in the *AWS Marketing page* and the *AWS Pricing Calculator*.

- To check AWS IoT Core pricing information, see AWS IoT Core Pricing.
- To estimate the cost of your architect solution, see AWS Pricing Calculator.