AWS IoT Things Graph

User Guide
AWS IoT Things Graph: User Guide
Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

- What is AWS IoT Things Graph? ................................................................. 1
- Model, flow, and mapping ..................................................................... 1
- How AWS IoT Things Graph works ......................................................... 2
  - Step 1: Build a flow .............................................................................. 2
  - Step 2: Create the flow configuration .................................................. 3
  - Step 3: Deploy ....................................................................................... 5
- How do I get started? .............................................................................. 5
- How a flow works ................................................................................... 6
- How a flow configuration works ............................................................. 6
  - Contents of a flow configuration ......................................................... 7
  - How to create a flow configuration ....................................................... 8
- Namespaces ........................................................................................... 8
- Getting started ....................................................................................... 9
  - Getting started with AWS IoT Things Graph in the cloud ................. 9
    - Prepare for cloud deployments ......................................................... 9
    - Cloud flow examples ...................................................................... 10
  - Getting started with AWS IoT Things Graph in AWS IoT Greengrass .. 20
    - Setting up your environment for AWS IoT Greengrass deployments .. 21
    - AWS IoT Greengrass flow examples ............................................... 22
- Using the example mock devices ......................................................... 52
  - What the mock devices do ................................................................. 52
  - Motion sensor .................................................................................... 53
  - Camera ............................................................................................... 54
  - Screen ................................................................................................. 54
  - Adapting the mock devices ............................................................... 55
- Modeling entities .................................................................................... 57
  - Modeling by device communication protocol .................................... 57
    - MQTT ............................................................................................... 57
    - Modbus ........................................................................................... 59
  - MQTT device modeling 101 ............................................................... 62
    - Creating your abstract device model .............................................. 63
    - Creating your device's capability (motion sensor) ......................... 64
    - Creating your device's capability (camera) .................................... 65
    - Creating your device definitions .................................................... 66
  - Modbus device modeling 101 .............................................................. 68
    - Creating your abstract device model .............................................. 69
    - Creating the definition for your device .......................................... 72
  - Service modeling 101 ......................................................................... 73
    - Modeling an AWS service ............................................................. 74
    - Modeling an AWS Lambda function ............................................. 77
- Creating and uploading entities ........................................................... 80
  - Step 1. Define the entities ................................................................. 80
  - Step 2. Upload the entity definitions .................................................. 82
  - Step 3. Associate devices with things ................................................ 86
- Versioning and entity modeling .......................................................... 88
  - Uploading entities ............................................................................. 88
  - Synchronizing with the public namespace ........................................ 88
- Viewing models and things in the AWS IoT Things Graph console ... 88
  - Models list ........................................................................................ 89
  - Model details ...................................................................................... 89
  - Model definition ................................................................................ 89
  - Associating things with devices ....................................................... 90
- Creating and deploying flows .............................................................. 91
- Creating and deploying flows (AWS IoT Greengrass) ......................... 91
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create and deploy a flow (CLI)</td>
<td>92</td>
</tr>
<tr>
<td>Create and deploy a flow (AWS IoT Things Graph console)</td>
<td>95</td>
</tr>
<tr>
<td>Creating and deploying flows (cloud)</td>
<td>99</td>
</tr>
<tr>
<td>Create and deploy a flow (CLI)</td>
<td>100</td>
</tr>
<tr>
<td>Create and deploy a flow (AWS IoT Things Graph console)</td>
<td>103</td>
</tr>
<tr>
<td>Revising flows</td>
<td>107</td>
</tr>
<tr>
<td>Adding triggers to flows</td>
<td>108</td>
</tr>
<tr>
<td>Time-based triggers</td>
<td>108</td>
</tr>
<tr>
<td>Device triggers</td>
<td>109</td>
</tr>
<tr>
<td>Triggers and flow configurations</td>
<td>110</td>
</tr>
<tr>
<td>Working with choice nodes</td>
<td>110</td>
</tr>
<tr>
<td>Create the flow</td>
<td>111</td>
</tr>
<tr>
<td>Configure events and choice node logic</td>
<td>113</td>
</tr>
<tr>
<td>Using conditional logic in choice nodes</td>
<td>120</td>
</tr>
<tr>
<td>Creating systems</td>
<td>125</td>
</tr>
<tr>
<td>Creating flow configurations (AWS IoT Greengrass)</td>
<td>126</td>
</tr>
<tr>
<td>Creating a flow configuration in the AWS CLI</td>
<td>126</td>
</tr>
<tr>
<td>Creating a flow configuration in the AWS IoT Things Graph console</td>
<td>129</td>
</tr>
<tr>
<td>Creating flow configurations (cloud)</td>
<td>133</td>
</tr>
<tr>
<td>Creating a flow configuration in the AWS CLI</td>
<td>134</td>
</tr>
<tr>
<td>Creating a flow configuration in the AWS IoT Things Graph console</td>
<td>136</td>
</tr>
<tr>
<td>Viewing flow configurations and flow executions in the AWS IoT Things Graph console</td>
<td>141</td>
</tr>
<tr>
<td>Flow configurations</td>
<td>141</td>
</tr>
<tr>
<td>Flow executions</td>
<td>142</td>
</tr>
<tr>
<td>Managing lifecycles for AWS IoT Things Graph entities, flows, systems, and deployments</td>
<td>144</td>
</tr>
<tr>
<td>Namespace management</td>
<td>144</td>
</tr>
<tr>
<td>Deleting systems, flows, and namespaces</td>
<td>145</td>
</tr>
<tr>
<td>Deleting flow configurations</td>
<td>146</td>
</tr>
<tr>
<td>Security</td>
<td>147</td>
</tr>
<tr>
<td>Security in AWS IoT Things Graph</td>
<td>147</td>
</tr>
<tr>
<td>IAM role for flow execution in cloud deployments</td>
<td>148</td>
</tr>
<tr>
<td>IAM role for flow execution in AWS IoT Greengrass deployments</td>
<td>148</td>
</tr>
<tr>
<td>Security in AWS IoT Greengrass deployments</td>
<td>149</td>
</tr>
<tr>
<td>MQTT subscriptions</td>
<td>150</td>
</tr>
<tr>
<td>Security in AWS IoT Things Graph namespaces</td>
<td>150</td>
</tr>
<tr>
<td>Data protection</td>
<td>151</td>
</tr>
<tr>
<td>Data encryption</td>
<td>151</td>
</tr>
<tr>
<td>Encryption at Rest</td>
<td>151</td>
</tr>
<tr>
<td>Encryption in transit</td>
<td>152</td>
</tr>
<tr>
<td>Key management</td>
<td>152</td>
</tr>
<tr>
<td>Internetwork traffic privacy</td>
<td>152</td>
</tr>
<tr>
<td>Data retention</td>
<td>152</td>
</tr>
<tr>
<td>Identity and access management</td>
<td>153</td>
</tr>
<tr>
<td>Audience</td>
<td>153</td>
</tr>
<tr>
<td>Authenticating with identities</td>
<td>153</td>
</tr>
<tr>
<td>Managing access using policies</td>
<td>155</td>
</tr>
<tr>
<td>How AWS IoT Things Graph works with IAM</td>
<td>156</td>
</tr>
<tr>
<td>Identity-based policy examples</td>
<td>160</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>163</td>
</tr>
<tr>
<td>Monitoring AWS IoT Things Graph</td>
<td>165</td>
</tr>
<tr>
<td>Monitoring with CloudWatch</td>
<td>166</td>
</tr>
<tr>
<td>Logging AWS IoT Things Graph API calls with AWS CloudTrail</td>
<td>167</td>
</tr>
<tr>
<td>Compliance validation</td>
<td>174</td>
</tr>
<tr>
<td>Resilience</td>
<td>174</td>
</tr>
<tr>
<td>Infrastructure security</td>
<td>175</td>
</tr>
<tr>
<td>Configuration and vulnerability analysis</td>
<td>175</td>
</tr>
</tbody>
</table>
What is AWS IoT Things Graph?

AWS IoT Things Graph is an orchestration service that simplifies development of IoT applications. These applications can use different devices and web services from different manufacturers that otherwise can't speak with each other because they use different protocols, data formats, and message syntaxes.

For example, a home security application can contain cameras, door and window sensors, and motion sensors from a variety of manufacturers that might not conform to the same IoT standards. As a result, these cameras and sensors aren't necessarily designed to communicate with each other. The application builder has to expend significant development effort to translate messages and actions from one device to another. This drives up software development costs and increases the time to market.

Additionally, the device vendor used might be different for each installation. Each time the device vendor changes, the development effort needs to be repeated. Customers prefer to have a large catalog of supported devices so that they can pick the most cost-effective alternative.

Lastly, in many sites, the flow may use web services and run in the cloud. In other sites, the automation flows need to be deployed on the edge gateways (such as AWS IoT Greengrass). Designing flows that can run either in the cloud or on edge gateways is difficult. AWS IoT Things Graph manages the interactions between the devices, ensuring that the necessary transformations take place.

Communicating through Models

AWS IoT Things Graph can communicate with the different devices and web services and orchestrate interactions between them through the use of reusable abstractions known as models. Models define the supported actions and events generated by the devices. They also describe how to invoke those actions and read the generated events. With AWS IoT Things Graph, application builders need to specify only the sequence of interactions in their application using models and the corresponding devices and web services. AWS IoT Things Graph uses the models to identify the protocols, addressing schemes, and data formats that the different devices use. It does all the necessary translating of messages between the devices and web services. This enables the flow to execute in the expected order.

Flow Design Interface

AWS IoT Things Graph offers a visual drag-and-drop flow design interface that drastically reduces the application development time. Application builders can define interactions graphically, and build their applications using just a few actions. The visual interface includes models that implement conditional operations such as if-then-else and logical operators such as AND/OR. This enables you to easily create multistep automation flows. Also, AWS IoT Things Graph enables you to replace devices in the system with newer versions as they become available, without doing additional work to integrate them into your application.

Prebuilt Models

The AWS IoT Things Graph console contains prebuilt models for commonly used devices, such as cameras and motion sensors, and web services, such as Amazon Rekognition and AWS Lambda. Application builders can also use AWS IoT Things Graph to create their own models for their devices. After you build an application, you can deploy your it to the AWS Cloud to orchestrate interactions between web services and devices connected to the AWS IoT cloud. You can also deploy it to an AWS IoT Greengrass-enabled device to orchestrate interactions between locally connected devices and web services.

Model, flow, and mapping

There are three key concepts in AWS IoT Things Graph: model, mapping, and flow.
Device and Service Model

Device and service models are representations of the agents that interact in a flow. Device and service models generically define the attributes, inputs, and outputs of the devices and services that interact with each other in flows. AWS IoT Things Graph enables you to define device and service interactions without considering the low-level implementations of devices produced by different manufacturers.

Flow

A flow (or workflow) consists of device models and services. Flows define how the devices and services interact with each other after a triggering event occurs. A flow lists these interactions as a sequence of steps. Each step contains an action on a device or web service and the related inputs to and outputs from that action. The flow defines the logical interactions and orders of execution between the devices and services.

For more information, see How a flow works.

Mapping

A mapping provides information that enables AWS IoT Things Graph to convert the output message of one device into the expected format for the next device in the flow. Mappings bridge differences between the devices and enable them to work together. Mappings enable you to build IoT applications that use a variety of devices from different manufacturers.

How AWS IoT Things Graph works

Follow these steps to create and deploy an AWS IoT Things Graph flow.

1. Build a Flow
2. Create/Configure Deployment
3. Deploy

Step 1: Build a flow

Build your flow by dragging device and web service models from the model library into a flow designer. (You can also find this page by choosing the menu icon at the upper left of the page, and then choosing Flows.)

The model library contains models that AWS IoT Things Graph has made publicly available (public models), and the models that you created (private models). Private models are visible only to you. Define interactions by connecting the model outputs with appropriate inputs of other models. Choose Publish to save your flow and publish it to the list of flows that can be deployed.
Step 2: Create the flow configuration

On the following pages that appear, you'll configure the deployment target (cloud or Greengrass) and map things in your registry to the devices in your flow.

Configure the target

When you create your flow configuration, do the following:

- For cloud deployments, specify the flow action ARN. This is the ARN of the IAM role that AWS IoT Things Graph will use when the flow executes in the cloud.
- For AWS IoT Greengrass deployments, specify the AWS IoT Greengrass group where the application will run. Also specify the Amazon Simple Storage Service (Amazon S3) bucket to store the application configuration so that AWS IoT Greengrass can retrieve it when it starts.

Cloud
Map things

When you create your flow configuration, select things from your registry that correspond with each device model in your flow.
Step 3: Deploy

After you configure your deployment, deploy the application to the cloud or to AWS IoT Greengrass-enabled devices.

AWS IoT Things Graph packages up the flow and its dependencies and pushes them either to the cloud or to the AWS IoT Greengrass-enabled device. AWS IoT Things Graph then manages interactions between locally connected devices.

How do I get started?

Explore existing devices and device models in the AWS IoT Things Graph console to see if the ones you want to use in your flows are available. If the ones you want aren't available, you can build your own models by using the built-in model editor in the AWS IoT Things Graph console.

You define flows by chaining together models and web services. Then you deploy the flow.

For more information about how a flow works, see How a flow works. For examples that demonstrate how this works, see Getting started with AWS IoT Things Graph.
How a flow works

When you deploy a flow configuration, the AWS IoT Things Graph runtime is deployed to the cloud or to an AWS IoT Greengrass core device. The AWS IoT Things Graph runtime identifies trigger devices and messages and handles communications between the devices and web services to ensure that the flow executes in the expected order. You can create model and flow definitions using either the AWS IoT Things Graph console or the AWS IoT Things Graph APIs.

The following diagram shows how an AWS IoT Things Graph flow works when device and flow definitions are deployed and a flow is ready to run. This is a hypothetical automated door entry workflow for a building's security system, and is an example of a business process represented as a graph.

In the diagram, the badge reader is defined by Model A, the validation web service by Model B, the turnstile by Model C, and the camera by Model D. When a valid badge is scanned, the badge reader triggers an event that causes the next step to execute. This step is a validation web service. The step calls the validation API in the service. In this way, AWS IoT Things Graph coordinates all of the steps in the graph.

The nodes in the graph represent things (devices or web services), and the edges represent connections between pairs of nodes.

Each thing is described by a model (a device definition). The models define each thing as a set of inputs, outputs, and attributes. Models also expose interfaces for the rest of the application to use. Each step (node) in a workflow (flow) represents an action for a device, service, or built-in control flow action to take. A connection between two nodes represents a logical interaction or an order of execution between two nodes. A flow is one or more steps chained together, and represents business processes.

How a flow configuration works

An AWS IoT Things Graph flow configuration implements a workflow (flow) for a specific location or deployment. A flow configuration consists of a flow, associated triggers, and all the corresponding physical things that interact with each other in the flows.

The following diagram shows the elements of a flow configuration and how they relate to each other.
Contents of a flow configuration

A flow configuration contains the following elements.

Flow

A flow (or workflow) consists of device models and services. Flows define how the devices and services interact with each other after a triggering event occurs. A flow lists these interactions as a sequence of steps. Each step contains an action on a device or web service and the related inputs to and outputs from that action. The flow defines the logical interactions and orders of execution between the devices and services.

For more information, see How a flow works.

Device and service models

Device and service models are representations of the agents that interact in a flow. Device and service models generically define the attributes, inputs, and outputs of the devices and services that interact with each other in flows. AWS IoT Things Graph enables you to define device and service interactions without considering the low-level implementations of devices produced by different manufacturers.

Triggers

Triggers are the events that start flows. Triggers define the conditions (such as an interval of time passing, a sensor detecting motion, or a change in temperature) that cause a flow to start. You define
triggers for each flow when you create a flow configuration, so different triggers can start a given flow in different deployments.

**Things**

Things are the physical devices used in the flows. When you create a flow configuration, you associate specific things with the device models used in the flows. For example, if your flow contains a thermostat device model, your flow configuration will contain a thermostat from your AWS IoT things registry that is associated with the thermostat device model.

**How to create a flow configuration**

You can create flow configurations by writing the GraphQL directly and using the [AWS IoT Things Graph APIs](https://docs.aws.amazon.com/iotthingsgraph/latest/ug/data-modeling-overview.html). The AWS IoT Things Graph console provides an easy way to create, maintain, and deploy your flow configurations.

For detailed instructions on how to create and deploy flow configurations using both the AWS CLI and the AWS IoT Things Graph console, see [Creating and deploying flows](https://docs.aws.amazon.com/iotthingsgraph/latest/ug/data-modeling-overview.html).

**Namespaces**

A namespace is the repository for entities (for example, models) created using the AWS IoT Things Graph Data Modeling (TDM) language. These entities are associated with a specific account. The namespace is part of the URN for each entity.

When you upload TDM entities for the first time, AWS IoT Things Graph creates a namespace for you. The namespace consists of your AWS Region, your account ID, and a fixed suffix `default`. The pattern is `REGION/ACCOUNT ID/default`. For example, a TDM entity created in the `us-west-2` region looks like this: `us-west-2/012345678910/default`.

You can extend the path under `default` to organize your entities. For example, you could place your device definitions under `default/devices` and your property definitions under `default/properties`.


There are two kinds of namespaces: public and private. The namespace associated with your account is private. You create and upload your own entity models inside this namespace. The public namespace is maintained by AWS and contains the entity definitions that all accounts can access. Private namespaces track the public namespace, and you can synchronize your private namespace with a specific version of the public namespace.

For more information about namespaces, see:

- Versioning and entity modeling
- Namespace versioning and workflows
- Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments

Getting started

This section contains example AWS IoT Things Graph entity models, workflows (flows), and deployment configurations to get you started with end-to-end development using AWS IoT Things Graph.

When you work with AWS IoT Things Graph, you must deploy the service and all associated AWS and AWS IoT services (such as AWS IoT Device Management, Amazon Simple Storage Service (Amazon S3) buckets, and AWS Lambda functions) in the same AWS Region.

Topics

- Getting started with AWS IoT Things Graph in the cloud (p. 9)
- Getting started with AWS IoT Things Graph in AWS IoT Greengrass (p. 20)
- Using the example mock devices (p. 52)

Getting started with AWS IoT Things Graph in the cloud

This section contains instructions and provides examples to get you started with end-to-end development using AWS IoT Things Graph in the cloud.

Topics

- Prepare for cloud deployments (p. 9)
- Cloud flow examples (p. 10)

Prepare for cloud deployments

This topic explains how to allow AWS IoT Things Graph to assume an IAM role with the appropriate permissions when it executes your flows in the cloud.

Create and configure an IAM role for cloud deployments

Cloud deployments require you to use an IAM role to allow AWS IoT Things Graph to execute the flows in the deployments on your behalf. A role that a service assumes to perform actions on your behalf is called a service role. For more information about this kind of role, see Creating a role to delegate permissions to an AWS service.

The role you create should have policies that allow AWS IoT Things Graph to perform all of the actions in your workflow. Workflows that contain devices or device models that publish and subscribe to MQTT messages need permission for `Iot:Connect`, `Iot:DescribeEndpoint`, and `Iot:Publish`. The getting started examples in this section also need permission for `Lambda`. If you are using any other web service in your workflows, add the appropriate policies. For sample AWS IoT policies that assign MQTT publish and subscribe permissions, see Publish/Subscribe policy examples.

When you deploy AWS IoT Things Graph flows to the cloud, you need to specify the IAM role that AWS IoT Things Graph assumes when it executes the flow in the cloud. If your flow uses other services, then your role must also have the appropriate permissions for those services.
For information about securing AWS resources, see IAM best practices. For information about best practices for attaching policies to IAM roles, see Grant least privilege and Get started using permissions with AWS managed policies.

This must have a trust relationship with AWS IoT Things Graph. The following instructions describe how to create this role.

1. In the IAM console (https://console.aws.amazon.com/iam/), choose Roles, and then choose Create Role.
2. On the Choose the service that will use this role page, choose IoT Things Graph, and then choose Next: Permissions.
3. On the Attach permissions policies page, choose policies that contain permissions that your flows require. All flows require read-write permissions to AWS IoT, and the example flows in this section require read permission to AWS Lambda. Choose Next: Tags.
4. On the Add tags (optional) page, add optional tags that help you categorize your role. Choose Next: Review.
5. On the Create role page, enter a name for your role. Choose Create role.

After you create this role, copy the role ARN so that you can use it when you create cloud flow configurations.

Cloud flow examples

The examples in this section walk you through the steps to create and deploy AWS IoT Things Graph workflows (flows) to the cloud.

Topics

- Creating a flow in the cloud with devices (p. 10)
- Creating a flow with devices by using the AWS CLI (p. 18)

Creating a flow in the cloud with devices

This topic walks you through the steps to create and deploy a simple workflow (flow) that consists of three mock devices running in the cloud: a motion sensor, a camera, and a screen. The mock devices pretend to be devices running on a Raspberry Pi.

Note

The flow that you create in this example is the cloud version of the flow in Creating a flow with devices in AWS IoT Greengrass flow examples. This cloud example currently shows how to run the flow only with mock devices.

The flow in this example is triggered when the motion sensor detects motion. The motion sensor sends a message to the camera, which takes a picture and sends it to the screen for display.

Find the code for the mock devices in the CloudMockDevices.zip file. You can run these scripts on any Linux environment, such as a Raspberry Pi or an Amazon EC2 instance, on which Python 2.7 is installed.

Prerequisites

To create this example, you need the following AWS resources:

- An AWS account
• An IAM role that is configured according to the instructions in Prepare for cloud deployments

Create things

Open the AWS IoT console and create three things: one thing for your motion sensor, and two things for the camera and screen that are attached to your Raspberry Pi.

For instructions on how to create things in the registry, see Register a device in the registry. Be sure to create and activate certificates for each thing.

For more information about AWS IoT concepts, see What is AWS IoT?

Set up your mock devices

To run the workflow (flow) with mock devices, you need to copy the Python scripts in the CloudMockDevices.zip file to your Linux environment. These scripts mimic the actions of real devices.

1. Install the AWS IoT Device SDK for Python.
2. Download and extract the CloudMockDevices.zip file. The extracted directory contains three files named cloudms.py, cloudcamera.py, and cloudscreen.py.
3. Create folders named ms, camera, and screen.
4. Copy the cloudms.py file into the ms directory, the cloudcamera.py file into the camera directory, and the cloudscreen.py file into the screen directory.
5. Copy the certificate and private key for each of the three things that you created into the appropriate directory on your Linux environment. Also copy the root certificate authority (CA) for AWS IoT into each directory. For example, the motion sensor certificate and private key should be in the ms directory, with the root CA for AWS IoT and the cloudms.py file.

If you haven't created and activated certificates for your things, follow the steps in Register a device in the registry. Make sure to attach a policy to each certificate.

Create and publish the flow

To create this flow with the AWS CLI instead of the AWS IoT Things Graph console, follow the instructions in Creating a flow with devices by using the AWS CLI.

1. Open the AWS IoT Things Graph console, and then choose Create flow.

2. Create a flow.

   In the Flow configuration pane that appears, name your flow (such as SecurityFlow). Choose Create flow.

3. Add the device models to the flow.

   Search for the MotionSensor device model. Select the device model and drag it into the flow designer. Do the same for the Camera and Screen device models.
4. Connect the device models.

In the flow designer, select the edge of the MotionSensor device model and connect it to the Camera device model. Also connect the Camera device model to the Screen device model.

5. Update the motion sensor trigger.

In the trigger editor that appears in the right pane, for Condition, choose StateChanged. For Action, choose ThingsGraph.startFlow.

6. Update the camera device action.

   a. In the flow designer, select the Camera device model.
   b. In the action editor that appears in the right pane, choose No action configured.
   c. For Action, choose capture.
   d. Expand Output, and then enter cameraResult.

7. Update the screen device model action.

   a. In the flow designer, select the Screen device model.
   b. In the action editor that appears in the right pane, choose No action configured.
   c. For Action, choose display.
   d. Expand Inputs, and then choose Define Input.
   e. For Inputs, enter ${cameraResult.lastClickedImage}.

8. Publish the flow.

   Choose Publish at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed.

**Associate things to device models**

1. Select the menu icon at the upper left of the page. Choose Things.
On the Things page, choose the motion sensor thing that you created earlier. Then choose Associate with device.

2. On the Select device model page, choose HCSR501MotionSensor. Choose Associate. This step associates the HCSR501MotionSensor motion sensor thing in your registry with the motion sensor device model in your flow. The HCSR501MotionSensor device implements the capability of the motion sensor device model.
3. After you return to the Select device model page, refresh the page to verify that the motion sensor thing is associated with the HCSR501MotionSensor device. Repeat the previous two steps for the RaspberryPiCamera and RaspberryPiScreen devices.

Create and deploy the flow configuration

1. Create the flow configuration.

Select the menu icon at the upper left of the page, and then choose Flows to return to the Flows page. Select the box next to the flow that you just created, and then choose Create flow configuration.

2. Name the flow configuration.

A flow configuration contains the details that are specific to a single deployment. On the Describe flow configuration page, select your flow and enter a flow configuration name. The flow configuration name can't contain spaces.

Choose Cloud, and then choose Next.

3. Configure the target.

On the Configure target page, enter the ARN of the role that you created in Prepare for cloud deployments. Choose Next.
4. Select things for your deployment.

The Map Things page provides an interface for selecting the specific things to include in your deployment. The menus under each device model in your deployment contain all of the things that you associated with the device model. Because you're getting started, the menus for each device on this page will include only one thing (the thing that you've associated with each device model).

On the Map Things page, from the menu under the motionSensor device model, select the motion sensor thing that you created earlier. Select the camera and screen things for the Camera and Screen device models. Choose Next.

5. View the trigger.

On the Set up triggers page, the GraphQL that defines the motion event trigger appears in the editor. This GraphQL specifies the event that triggers the flow. When the motion sensor detects a person moving, the ThingsGraph.startFlow function initiates the flow. You don't need to edit this code.

Choose Review.
6. Review and create.

On the **Review and create** page, review the information you entered for your flow configuration. Then choose **Create**.

When the **Flow configuration created** message appears, choose **Deploy now**.

Refresh the **Deployments** page to verify that the flow has deployed. After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

**Run the flow with mock devices**

To run the Python scripts for each mock device, you need to know your AWS IoT custom endpoint. This enables you to connect to AWS IoT.

To find this endpoint, open the AWS IoT console ([https://console.aws.amazon.com/iot/](https://console.aws.amazon.com/iot/)), and then choose **Settings**. You can also find the endpoint by using the **aws iot describe-endpoint** AWS CLI command.
If you’re using an Amazon Trust Services (ATS) root CA, use the following AWS CLI command to find your custom endpoint. For more information about certificates and ATS endpoints, see X.509 certificates and AWS IoT.

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

Follow these steps to run all three Python scripts and observe the mock devices interacting in the flow.

1. Open three terminal sessions on your Linux environment. In one terminal, navigate to the ms directory. To run the `ms.py` script, run the following command.

```
python cloudms.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n motion sensor thing name -id "cloudms"
```

After the script starts running, it displays the following output every 10 seconds. This indicates that the mock motion sensor has detected motion.

```
Published topic TG_MS/motion: {"isMotionDetected": true}
```

2. In the second terminal, navigate to the camera directory. Run the `camera.py` script by using the following command.

```
python cloudcamera.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n camera thing name -id "cloudcamera"
```

After the script starts running, it displays the following output every 10 seconds. This indicates that the motion sensor trigger has prompted the mock camera to capture an image and publish it to the `/capture/finished` MQTT topic.

```
Received message on topic TG_Camera/capture: {}
2019-02-11 21:37:33,430 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
```

3. In the third terminal, navigate to the screen directory. Run the `screen.py` script by using the following command.

```
python cloudscreen.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n screen thing name -id "cloudscreen"
```

After the script starts running, it displays the following output every 10 seconds. This indicates that the mock screen has received the image from the mock camera.

```
Received message on topic TG_Screen/display: {"imageUri": "https://images-na.ssl-images-amazon.com/images/I/51rMLSWgwRL._AC_US218_.jpg"}
```
Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you've created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Creating a flow with devices by using the AWS CLI

This topic contains the AWS CLI commands that create the flow in Creating a flow with devices. The setup instructions for this example are identical to the ones in that topic. Before starting with this example, follow all of the instructions in that topic up to Create and deploy the flow, and then return to this topic.

Install the CLI

To install the AWS CLI, follow the instructions in Installing the AWS CLI.

Create the flow by using the AWS CLI

The following steps describe how to create and deploy the flow by using the AWS CLI.

1. The following AWS IoT Things Graph Data Model (TDM) code contains the definition of the flow used in this example.

   Copy this code to a file. Replace the `REGION` and `ACCOUNT_ID` placeholders with your AWS Region and account ID.

   ```
   { 
   query SecurityFlow($camera: string!, $screen: string!) @workflowType(id: 
   "urn:tdm:REGION/ACCOUNT_ID/default:Workflow:SecurityFlow") { 
   variables { 
   cameraResult @property(id: "urn:tdm:aws/examples:property:CameraStateProperty") 
   } 
   steps { 
   step(name: "Camera", outEvent: ["cameraStepDone"]) { 
   DeviceActivity(deviceModel: "urn:tdm:aws/examples:deviceModel:Camera", out: 
   "cameraResult", deviceId: "$\{camera\}") { 
   capture 
   } 
   } 
   step(name: "Screen", inEvent: ["cameraStepDone"]) { 
   DeviceActivity(deviceModel: "urn:tdm:aws/examples:deviceModel:Screen", 
   deviceId: "$\{screen\}") { 
   display(imageUrl: "$\{cameraResult.lastClickedImage\}"") 
   } 
   } 
   }
   }
   ```

2. Enter the following command to create the flow. This command assumes that you're working in a Linux or Unix environment. For other environments, use the equivalent of the `cat` utility.

   ```
   aws iotthingsgraph create-flow-template --definition 
   language=GRAPHQL,text=""\$(cat PATH TO TDM FILE)"
   ```
3. The following TDM code contains the definition of the system used in this example. The console creates this system for you.

Copy this code to a file. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID.

```json
{
  type SecurityFlow @systemType(id: \"urn:tdm:REGION/ACCOUNT ID/ default:System:SecurityFlow\", description: \"Home Security System\") {
    motionSensor: Device @thing(id: \"urn:tdm:aws/examples:deviceModel:MotionSensor\")
    camera: Device @thing(id: \"urn:tdm:aws/examples:deviceModel:Camera\")
    screen: Device @thing(id: \"urn:tdm:aws/examples:deviceModel:Screen\")
}
}
```

4. Enter the following command to create the system.

```bash
aws iotthingsgraph create-system-template --definition
language=GRAPHQL,text="""$(cat
PATH TO TDM FILE)"""
```

5. Enter the following commands to associate the things you created in your registry with the device models used in the flow.

Replace the `MotionSensorName`, `CameraName`, and `ScreenName` placeholders with the names of the things you created earlier.

```bash
aws iotthingsgraph associate-entity-to-thing --thing-name "MotionSensorName" --entity-id "urn:tdm:aws/examples:Device:HCSR501MotionSensor"
aws iotthingsgraph associate-entity-to-thing --thing-name "CameraName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiCamera"
aws iotthingsgraph associate-entity-to-thing --thing-name "ScreenName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiScreen"
```

6. The following TDM code contains the definition of the flow configuration used in this example. The TDM definition is inside the `definition` object. For more information, see Creating flow configurations.

Copy this code to a file. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID. Replace the `MotionSensorName`, `ScreenName`, and `CameraName` placeholders with the names of the things you created earlier.

```json
{
  query Room215 @deployment(id: \"urn:tdm:REGION/ACCOUNT ID/default:Deployment:Room215\", systemId: \"urn:tdm:REGION/ACCOUNT ID/default:System:SecurityFlow\") {
    motionSensor(deviceId: \"MotionSensorName\")
    screen(deviceId: \"ScreenName\")
    camera(deviceId: \"CameraName\")
    triggers {
      MotionEventTrigger(description: \"a trigger\") {
        condition[expr: \"devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent"]
        action[expr: \"ThingsGraph.startFlow('SecurityFlow', bindings[name == 'camera'].deviceId, bindings[name == 'screen'].deviceId)\"]
      }
    }
}
```
7. Enter the following command to create the flow configuration.

Replace `ROLE ARN` with the IAM role ARN that you created in Prepare for cloud deployments.

```
aws iotthingsgraph create-system-instance --definition 
  language=GRAPHQL,text="$(cat PATH TO TDM FILE)"
  --target CLOUD --flow-actions-role-arn ROLE ARN
```

When the operation completes, the AWS CLI returns the following deployment summary as a JSON object. Use the `id` value in the `summary` block as the TDM URN of the flow configuration.

```
{
  "summary": {
    "status": "PENDING_DEPLOYMENT",
    "greengrassGroupName": "",
    "target": "CLOUD",
    "arn": "arn:aws:iotthingsgraph:REGION:ACCOUNT ID:default#Deployment#Room215",
    "updatedAt": 1555021747.176,
    "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Room215",
    "createdAt": 1555021747.176
  }
}
```

8. Enter the following command to deploy the flow configuration to your AWS IoT Greengrass group. Use the TDM URN value returned in the previous step as the value of the `id` parameter.

```
aws iotthingsgraph deploy-system-instance --id SYSTEM INSTANCE URN
```

9. To verify the deployment, follow the steps in Run the Flow with Mock Devices (p. 16).

Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you’ve created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Getting started with AWS IoT Things Graph in AWS IoT Greengrass

This section contains instructions and provides examples to get you started with end-to-end development using AWS IoT Greengrass deployments of AWS IoT Things Graph.

Topics
- Setting up your environment for AWS IoT Greengrass deployments (p. 21)
- AWS IoT Greengrass flow examples (p. 22)
Setting up your environment for AWS IoT Greengrass deployments

This topic describes the tasks you need to do to start working with AWS IoT Things Graph. You can't deploy and run a flow successfully until you perform the these tasks.

Before you get started, make sure that you have an AWS account.

Create an AWS IoT Greengrass group

**Note**
For information about using AWS CloudFormation to create and manage AWS IoT Greengrass groups and resources, see [AWS IoT Greengrass resource types reference](https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-resource-types-reference.html).

To perform the tasks related to setting up your AWS IoT Greengrass core, you need a Mac, Windows, or Linux or Unix system. The setup instructions in [Getting started with AWS IoT Greengrass](https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-get-started.html) recommend that you start with a Raspberry Pi.

1. Download and install the AWS IoT Greengrass core software.
   
   Follow the instructions in [Installing AWS IoT Greengrass core software](https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-get-started.html).

2. Start the AWS IoT Greengrass software.
   
   Follow the instructions in [Deploy cloud configurations to an AWS IoT Greengrass core device](https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-get-started.html) to start AWS IoT Greengrass.

3. Create a directory named `thingsgraph` at the root directory of your AWS IoT Greengrass core device. AWS IoT Things Graph installs files in this location. This directory must have read, write, and execute permissions for `ggc_user`. If your AWS IoT Greengrass core is using a Unix-like operating system, set these permissions by entering the following commands at a command prompt.

   ```
sudo chown ggc_user: /thingsgraph
sudo chmod 700 /thingsgraph
   ```

   The AWS IoT Things Graph logs go into the `/greengrass/ggc/var/log/user/us-east-1/ThingsGraph/` directory. For information about how to configure AWS IoT Greengrass logs, see [Monitoring with AWS IoT Greengrass logs](https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-monitor-logs.html).

4. Create an IAM role for AWS IoT Greengrass and attach the policies that your flows will need to interact with other AWS services when they're running.

   For example, a flow that needs to read and write to an Amazon S3 bucket needs to have the `AmazonS3FullAccess` policy attached to it.

5. When AWS IoT Things Graph installs a flow to your AWS IoT Greengrass core, it first uploads a file that contains all of the flow's dependencies to an Amazon S3 bucket. It then installs the flow and its dependencies from the bucket. This means that the AWS IoT Greengrass service needs permission to access this bucket.

   In the IAM console (https://console.aws.amazon.com/iam/), choose **Roles**. Select the role named `Greengrass_ServiceRole`, and attach the `AmazonS3FullAccess` policy to give it access to your S3 bucket.

**Install Java**

To run AWS IoT Things Graph, your AWS IoT Greengrass core device needs to have Java 8 installed.
If you're using a Linux environment, follow these steps to install the correct version of Java. See the Java website for instructions on installing Java 8 on Windows and Mac environments.

1. Run the following command to install Java 8.
   ```bash
   sudo yum install java-1.8.0
   ```
2. Run the following command to uninstall Java 7.
   ```bash
   sudo yum remove java-1.7.0-openjdk
   ```
3. Run the following command to create a symlink to the newly installed version of Java.
   ```bash
   sudo ln /etc/alternatives/java /usr/bin/java8
   ```

AWS IoT Greengrass flow examples

The examples in this section walk you through the steps to create and deploy workflows (flows) to an AWS IoT Greengrass group.

Topics
- Creating a flow in an AWS IoT Greengrass group with devices (p. 22)
- Creating a flow with Lambda functions (p. 32)
- Creating a flow with devices and a service (p. 37)
- Creating a flow with devices by using the AWS CLI (p. 44)
- Creating a flow with Lambda functions by using the AWS CLI (p. 46)
- Creating a flow with devices and a service by using the CLI (p. 49)

Creating a flow in an AWS IoT Greengrass group with devices

This topic walks you through the steps to create and deploy a simple workflow (flow) that consists of a Raspberry Pi connected to three devices: an Aukru HC-SR501 motion sensor, a Raspberry Pi camera, and a Raspberry Pi screen.

The flow in this example is triggered when the motion sensor detects motion. The motion sensor sends a message to the camera, which takes a picture and sends it to the screen for display.

If you don't want to buy the devices used in this example, you can run three Python scripts that mimic the functions of the devices. This topic shows you how to run the flow with both the real and mock devices.

Find the code for the mock devices in the MockDevices.zip file. You can run these scripts on any Linux environment, such as a Raspberry Pi or an Amazon EC2 instance, on which Python 2.7 is installed.

Prerequisites

To create this example, you need the following AWS resources:

- An AWS account
- An Amazon S3 bucket

To create this example with real devices, you need the following hardware resources:

- A Raspberry Pi running AWS IoT Greengrass version 1.7 or later
- An Aukru HC-SR501 motion sensor
- A Raspberry Pi 7” touch screen display
• A Raspberry Pi camera module v2-8 megapixel, 1080p

Set up your Raspberry Pi

To run the workflow with real devices, you need to set up your Raspberry Pi with those devices. If you’re creating this example with the mock devices, skip ahead to Create an AWS IoT Greengrass Group (p. 23).

• Attach your motion sensor to Raspberry Pi GPIO pin 1.
• Attach the camera by following the instructions in the Raspberry Pi camera board documentation.
• Attach the screen by following the instructions in the Raspberry Pi 7” touch screen assembly guide.
• Download and install the drivers included in the ThingsGraphPrototypeDevices.zip file. To install the drivers, follow the instructions in the README.

Now you’ve configured your Raspberry Pi to communicate with the motion sensor, camera, and screen.

Create an AWS IoT Greengrass group

Note
For information about using AWS CloudFormation to create and manage AWS IoT Greengrass groups and resources, see AWS IoT Greengrass resource types reference.

If you’re creating this example with real devices, install AWS IoT Greengrass on the same Raspberry Pi to which you attached your motion sensor, camera, and screen.

If you’re using mock devices, install AWS IoT Greengrass on the same Linux environment where you run the Python scripts.

Perform the steps in Setting up your environment on the appropriate device or computer. Make sure that you’ve created an IAM role for AWS IoT Greengrass (step 4 in the Create an AWS IoT Greengrass Group procedure in that topic). Attach the AmazonS3FullAccess policy to give it access to your S3 bucket.

Note
The AWS IoT Greengrass group and Amazon S3 bucket that you use in this example must be created in the same AWS Region. The AWS IoT Things Graph entities that you create must also be in the same Region as these resources.

Create things

Open the AWS IoT console and create three things: one thing for your motion sensor, and two things for the camera and screen that are attached to your Raspberry Pi.

For instructions on how to create things in the registry, see Register a device in the registry. Be sure to create and activate certificates for each thing.

In the AWS IoT console, on the Greengrass tab, select your group, choose Devices, and then add the three new things to your AWS IoT Greengrass group.

For more information about AWS IoT concepts, see What is AWS IoT?

Set up your mock devices

To run the workflow (flow) with mock devices, you need to install the AWS IoT Greengrass Core SDK, and then copy the Python scripts in the MockDevices.zip file to your Linux environment.

If you’re creating this example with real devices, skip ahead to Create and Deploy the Flow (p. 27).

1. To download and decompress the AWS IoT Greengrass Core SDK, follow the steps in Create and package a Lambda function. (Choose the Unix-like system tab on the step for decompressing the greengrass-core-python-sdk-1.3.0.tar.gz file.)
You don't need to create the greengrassHelloWorld AWS Lambda function. Stop when you've extracted the python_sdk_1_3_0.zip file.

2. Download and extract the MockDevices.zip file. The extracted directory contains three files named ms.py, camera.py, and screen.py.

3. Create folders named ms, camera, and screen.

4. Copy the ms.py file into the ms directory, the camera.py file into the camera directory, and the screen.py file into the screen directory.

5. Copy the greengrasssdk directory (in sdk\python_sdk_1_3_0) into each of the mock device directories that you created.

6. Copy the certificate and private key for each of the three things that you created into the appropriate directory on your Linux environment. Also copy the root certificate authority (CA) for AWS IoT into each directory. For example, the motion sensor certificate and private key should be in the ms directory, with the root CA for AWS IoT and the ms.py file.

If you haven't created and activated certificates for your things, follow the steps in Register a device in the registry. Make sure to attach a policy to each certificate.

Create and publish the flow

To create this flow with the AWS CLI instead of the AWS IoT Things Graph console, follow the instructions in Creating a flow with devices by using the AWS CLI.

1. Open the AWS IoT Things Graph console, and then choose Create flow.

2. Create a flow.

   In the Flow configuration pane that appears, name your flow (such as SecurityFlow). Choose Create flow.

3. Add the device models to the flow.

   Search for the MotionSensor device mode. Select the device model and drag it into the flow designer. Do the same for the Camera and Screen device models.

4. Connect the device models.

   In the flow designer, select the edge of the MotionSensor device model and connect it to the Camera device model. Also connect the Camera device model to the Screen device model.
5. Update the motion sensor trigger.

   In the trigger editor that appears in the right pane, for **Condition**, choose **StateChanged**. For **Action**, choose **ThingsGraph.startFlow**.

6. Update the camera device module action.
   a. In the flow designer, select the **Camera** device model.
   b. In the action editor that appears in the right pane, choose **No action configured**.
   c. For **Action**, choose **capture**.
   d. Expand **Output**, and then enter **cameraResult**.

7. Update the screen device action.
   a. In the flow designer, select the **Screen** device model.
   b. In the action editor that appears in the right pane, choose **No action configured**.
   c. For **Action**, choose **display**.
   d. Expand **Inputs**, and then choose **Define Input**.
   e. For **Inputs**, enter **${cameraResult.lastClickedImage}**.

8. Publish the flow.

   Choose **Publish** at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed.

**Associate things to device models**

1. Select the menu icon at the upper left of the page. Choose **Things**.
On the Things page, choose the motion sensor thing that you created earlier. Then choose **Associate with device**.

2. On the Select device model page, choose **HCSR501MotionSensor**. Choose **Associate**. This step associates the HCSR501MotionSensor motion sensor thing in your registry with the motion sensor device model in your flow. The HCSR501MotionSensor device implements the capability of the motion sensor device model.
3. After you return to the Select device model page, refresh the page to verify that the motion sensor thing is associate with the HCSR501MotionSensor device. Repeat the previous two steps for the RaspberryPiCamera and RaspberryPiScreen devices.

Create and deploy the flow configuration

1. Create the flow configuration.

   Select the menu icon at the upper left of the page, and then choose Flows to return to the Flows page. Select the box next to the flow that you just created, and then choose Create flow configuration.

2. Name the flow configuration.

   A flow configuration contains the details that are specific to a single deployment. On the Describe flow configuration page, select your flow and enter a flow configuration name. The flow configuration name can't contain spaces.

   Choose Greengrass, and then choose Next.

3. Configure the target.

   On the Configure target page, enter the name of your Amazon S3 bucket and the AWS IoT Greengrass group to which your AWS IoT Greengrass core device belongs. Amazon S3 buckets are globally unique, so your bucket name will be different from the one in the following screen shot. Choose Next.
4. Select things for your deployment.

The Map Things page provides an interface for selecting the specific things to include in your deployment. The menus under each device model in your deployment contain all of the things that you associated with the device model. Because you're getting started, the menus for each device model on this page will include only one thing (the thing that you've associated with each device model).

On the Map Things page, for motionSensor, select the motion sensor thing that you created earlier. Select the screen and camera things for the Camera and Screen device models. Choose Next.

5. View the trigger.

On the Set up triggers page, the GraphQL that defines the motion event trigger appears in the editor. This GraphQL specifies the event that triggers the flow. When the motion sensor detects a person moving, the ThingsGraph.startFlow function initiates the flow. You don't need to edit this code.

Choose Review.
6. Review and create.

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.

7. Deploy.

When the **Flow configuration created** message appears, choose **Deploy now**.

Refresh the **Deployments** page to verify that the flow has deployed. After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

**Run the flow with real devices**

If you created this example with mock devices, skip ahead to **Run the Flow with Mock Devices (p. 30)**.

Turn on your Raspberry Pi, motion sensor, camera, and screen. Position the motion sensor and camera so that they are directly in front of you. Wave your hand over the motion sensor. The camera takes a picture of you and displays it on the screen.
Run the flow with mock devices

To run the Python scripts for each mock device, you need to know your AWS IoT custom endpoint. This enables you to connect to AWS IoT.

To find this endpoint, open the AWS IoT console (https://console.aws.amazon.com/iot/), and then choose Settings. You can also find the endpoint by using the `aws iot describe-endpoint` AWS CLI command.

If you're using an Amazon Trust Services (ATS) root CA, use the following AWS CLI command to find your custom endpoint. For more information about certificates and ATS endpoints, see X.509 certificates and AWS IoT.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```
Follow these steps to run all three Python scripts and observe the mock devices interacting in the flow.

1. Open three terminal sessions on your Linux environment. In one terminal, navigate to the `ms` directory. To run the `ms.py` script, run the following command.

   ```bash
   python ms.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n motion sensor thing name
   ```

   After the script starts running, it displays the following output every 10 seconds. This indicates that the mock motion sensor has detected motion.

   ```json
   Published topic motion sensor thing name/motion: {"isMotionDetected": true}
   ```

2. In the second terminal, navigate to the `camera` directory. Run the `camera.py` script by using the following command.

   ```bash
   python camera.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n camera thing name
   ```

   After the script starts running, it displays the following output every 10 seconds. This indicates that the motion sensor trigger has prompted the mock camera to capture an image and publish it to the `/capture/finished` MQTT topic.

   ```json
   Received message on topic camera thing name/capture: {}
   2019-02-11 21:37:33,430 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
   ```

3. In the third terminal, navigate to the `screen` directory. Run the `screen.py` script by using the following command.

   ```bash
   python screen.py -e custom endpoint -r root CA file path -c certificate file path -k private key file path -n screen thing name
   ```

   After the script starts running, it displays the following output every 10 seconds. This indicates that the mock screen has received the image from the mock camera.

   ```json
   Received message on topic screen thing name/display: {"imageUri":"https://images-na.ssl-images-amazon.com/images/I/51rMLSWgwRL._AC_US218_.jpg"}
   ```

Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you’ve created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.
Creating a flow with Lambda functions

This topic describes how to create a simple workflow (flow) that uses three AWS Lambda functions as service models in that flow.

The flow runs every 60 seconds and calls three service models: one that reads data from an Amazon Simple Storage Service (Amazon S3) bucket, one that analyzes the data, and one that saves the results of the analysis to the Amazon S3 bucket.

Prerequisites

To create this example, you need the following resources.

- An AWS account.
- An AWS IoT Greengrass core, version 1.7 or later.
- An AWS IoT Greengrass group.
- Node.js installed on your AWS IoT Greengrass core device.
- An Amazon S3 bucket that contains a file named HelloWorld.txt.
- An AWS IoT Greengrass IAM role that has access to your S3 bucket.

Add this role to your AWS IoT Greengrass group. For information about how to configure IAM roles for AWS IoT Greengrass, see Configure IAM roles.

- A Lambda IAM role that has access to your S3 bucket.

Create an AWS IoT Greengrass group

Note
For information about using AWS CloudFormation to create and manage AWS IoT Greengrass groups and resources, see AWS IoT Greengrass resource types reference.

Perform the steps in Setting up your environment. Make sure that you've created an IAM role for AWS IoT Greengrass (step 4 in the Create an AWS IoT Greengrass Group procedure in that topic). Attach the AmazonS3FullAccess policy to give it access to your S3 bucket.

On the Settings page of your group, make sure that Greengrass container is selected under Lambda runtime environment.

Create and deploy the Lambda functions

1. Create the Lambda functions.
   1. Follow the instructions in Create a simple Lambda function to create three Lambda functions named GetS3Object, WordCount, and SaveToS3.

      Note
      The sample won't work if you use any other function names.

   2. The code for these functions is in the Lambdas.zip file. Don't add any triggers. Choose the latest supported Node.js runtime. Use an IAM role that has read/write access to your Amazon S3 bucket.

      In the SaveToS3 function, replace the value of bucket with the name of your S3 bucket.

      Each Lambda function is a service model in the flow. Copy the Lambda Amazon Resource Names (ARNs) so that you can use them in your service model definitions.
3. After you copy and save the functions in the Lambda console, on the **Actions** menu, choose **Publish new version**. You must use version 1 for each Lambda function.

2. Deploy the Lambda functions to AWS IoT Greengrass.

   To do this, follow the instructions in Configure the Lambda function for AWS IoT Greengrass.

### Create and publish the flow

To create this flow with the AWS CLI instead of the AWS IoT Things Graph console, follow the instructions in Creating a flow with Lambda functions by using the AWS CLI.

1. Open the AWS IoT Things Graph console.

   Choose **Create flow**.

   ![Create new flow](image)

2. Create a flow.

   In the **Flow configuration** pane that appears, enter a name for your flow. Choose **Create flow**.

3. Add the service models to the flow.

   On the **Logic** tab, choose **Clock**, and then drag it into the flow designer.

   On the **Service** tab, search for the **getS3Lambda** service model. Choose the service model and drag it into the flow designer. Do the same for the **wordCountLambda** and **saveResponseLambda** service models.

4. Connect the service models.

   In the flow designer, select the edge of the **ClockTrigger** service model and connect it to the **getS3Lambda** service model. Then connect the **getS3Lambda**, **wordCountLambda**, and **saveResponseLambda** service models in the same way.

5. Update the **ClockTrigger**.

---

33
In the trigger editor that appears in the right pane, for **Frequency**, enter 60, and then select **seconds** from the menu on the right. For **Action**, choose **ThingsGraph.startFlow**.

6. Add the `getS3Lambda` service model action.
   a. In the flow designer, choose the `getS3Lambda` service model. In the action editor that appears in the right pane, select **No action configured**. In the list that appears, select `getS3ObjectAsString`.
   b. Expand **Inputs**. Choose **Define Input**, and then enter the names of your Amazon S3 bucket and file as the values for **bucket** and **key**.
   c. Expand **Output**, and then enter `getS3LambdaResult`.

7. Add the `wordCountLambda` service model action.
   a. In the flow designer, select the `wordCountLambda` service model. In the action editor that appears in the right pane, choose **No action configured**. For **Action** box, choose **wordCount**. Expand **Inputs**, choose **Define Input**, and then for **message**, enter `${getS3LambdaResult.message}`.
   b. Expand **Output**, and then enter `wordCountLambdaResult`.

8. Add the `saveResponseLambda` service model action.
   a. In the flow designer, select the `saveResponseLambda` service model.
   b. In the action editor that appears in the right pane, choose **No action configured**.
   c. For **Action**, choose **save**.
   d. Expand **Inputs**, choose **Define Input**, and then for **response**, enter `$(wordCountLambdaResult)`.

9. Publish the flow.
   Choose **Publish** at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed.

**Create and deploy the flow configuration**

1. On the **Flows** list page, select the box next to the flow that you just created, and then choose **Create flow configuration**.

![Flow list page](image)

2. Name the flow configuration.
   On the **Describe flow configuration** page, select your flow and enter a flow configuration name. The flow configuration name can't contain spaces. Choose **Greengrass**, and then choose **Next**.

3. Configure the target.
On the **Configure target** page, enter the name of your Amazon S3 bucket and the AWS IoT Greengrass group to which your AWS IoT Greengrass core device belongs. Amazon S3 buckets are globally unique, so your bucket name will be different from the one in the following screen shot. Choose **Next**.

![Configure target](image)

4. Select things.

This example contains only service models, so you don't have to select any things to use in the flow. Choose **Next**.

5. View the trigger.

On the **Define trigger** page, the following GraphQL appears in the editor. This GraphQL specifies the time intervals at which the flow runs. This flow runs every 60 seconds. You don't need to edit this code.

Choose **Review**.
6. Review and create.

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.

7. Deploy.

When the **Flow configuration created** message appears, choose **Deploy now**.

Refresh the **Deployments** page to verify that the flow has deployed. After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

**Run the flow**

After a flow is deployed, it runs automatically every 60 seconds. You can verify that it's running by going to your Amazon S3 bucket. You should see a new file named `word-count-response` in the bucket.
Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you've created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Creating a flow with devices and a service

This topic walks you through the steps to create and deploy a simple workflow (flow) that consists of a Raspberry Pi connected to three devices: an Aukru HC-SR501 motion sensor, a Raspberry Pi camera, and a Raspberry Pi screen. It also contains an AWS IoT Things Graph service model that exposes the face detection capability of Amazon Rekognition.

The flow is triggered when the motion detector detects a person moving. The motion detector sends a message to the camera, which takes a picture and sends it to the Amazon Rekognition service model. The service model sends the image to display to the screen. The service model doesn't process the results of the DetectFaces API or send anything other than the image to the screen.

You can modify this example by adding a service model between the Amazon Rekognition service and the screen. The additional service model can process the results (such as determining whether the image should be displayed on the screen) before sending the image to the screen.

Prerequisites

To create this example, you need the following resources:

- An AWS account
- A Raspberry Pi running AWS IoT Greengrass version 1.7 or later
- An Aukru HC-SR501 motion sensor
- A Raspberry Pi 7" touchscreen display
- A Raspberry Pi camera module v2-8 megapixel, 1080p
- An Amazon S3 bucket

Set up your Raspberry Pi

To run the workflow (flow), you need to set up your Raspberry Pi.

- Attach your motion sensor to Raspberry Pi GPIO pin 1.
- Attach the camera by following the instructions in the Raspberry Pi camera board documentation. Attach the screen by following the instructions in the Raspberry Pi 7" touch screen assembly guide.
- Download and install the drivers included in the ThingsGraphPrototypeDevices.zip file. Follow the instructions in the README to install the drivers.

Now you've configured your Raspberry Pi to communicate with the motion sensor, camera, and screen.

Create an AWS IoT Greengrass group

Note
For information about using AWS CloudFormation to create and manage AWS IoT Greengrass groups and resources, see AWS IoT Greengrass resource types reference.

Install AWS IoT Greengrass on the same Raspberry Pi to which you attached your motion sensor, camera, and screen.

Perform the steps in Setting up your environment on your Raspberry Pi. Make sure that you've created an IAM role for AWS IoT Greengrass (step 4 in the Create an AWS IoT Greengrass group procedure in that topic). Attach the AmazonS3FullAccess policy to give it access to your S3 bucket.
Note
The AWS IoT Greengrass group and Amazon S3 bucket that you use in this example must be created in the same AWS Region. The AWS IoT Things Graph entities that you create must also be in the same Region as these resources.

Create things
Open the AWS IoT console and create three things: one thing for your motion sensor, and two things for the camera and screen that are attached to your Raspberry Pi.

For instructions on how to create things in the registry, see Register a device in the registry.

On the Greengrass tab in the AWS IoT console, choose your group, choose Devices, and then add the three new things to your AWS IoT Greengrass group.

For general information about AWS IoT concepts, see What is AWS IoT?

Create and publish the flow
To create this flow with the AWS CLI instead of the AWS IoT Things Graph console, follow the instructions in Creating a flow with devices and a service by using the AWS CLI.

1. Open the AWS IoT Things Graph console.

2. Create a flow.

   In the Flow configuration pane that appears, name your flow (such as RekognitionFlow). Choose Create flow.

3. Add the device models and service model to the flow.

   1. Search for the MotionSensor device model. Select the device and drag it into the flow designer.
   2. Search for the CameraRkgnExample device model. Select the device model and drag it into the flow designer.
   3. On the Services tab, search for the Rekognition service model. Select the service model and drag it into the flow designer.
   4. On the Devices tab, do the same for the Screen device model.
4. Connect the device models.

In the flow designer, select the edge of the MotionSensor device model and connect it to the CameraRkgnExample device model. Then connect the CameraRkgnExample device model, Rekognition service model, and the Screen device model in the same way.

5. Update the motion sensor trigger.

In the trigger editor that appears in the right pane, for Condition, choose StateChanged. For Action, choose ThingsGraph.startFlow.

6. Update the camera device model action.

a. In the flow designer, select the CameraRkgnExample device model.
   b. In the action editor that appears in the right pane, select No action configured.
   c. For Action, choose capture.
   d. Expand Output, and then enter cameraRkgnExampleResult.

7. Update the Rekognition service model activity.

a. In the flow designer, select the Rekognition service model.
   b. In the action editor that appears in the right pane, select No action configured.
   c. For Action, choose detectFaces.
   d. Expand Inputs, and then choose Define Input.
   e. For bucketName, enter ${cameraRkgnExampleResult.s3BucketName}.
   f. For itemName, enter ${cameraRkgnExampleResult.s3ItemName}.
   g. Expand Output, and then enter rekognitionResult.

8. Update the screen device model activity.

a. In the flow designer, select the Screen device model.
   b. In the action editor that appears in the right pane, select No action configured.
   c. For Action, choose display.
   d. Expand Inputs, and then choose Define Input.
   e. For imageUrl, enter ${cameraRkgnExampleResult.lastClickedImage}.

9. Publish the flow.

Choose Publish at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed.

**Associate things to device models**

1. Associate things to device models.

   Select the menu icon at the upper left of the page. Then choose Things.
On the **Things** page, select the motion sensor thing that you created earlier. Then choose **Associate with device**.

2. On the **Select device model** page, choose **HCSR501MotionSensor**. Choose **Associate**. This step associates the HCSR501MotionSensor motion sensor thing in your registry with the motion sensor device model in your flow. The HCSR501MotionSensor device implements the capability of the motion sensor device model.
3. After you return to the Select device model page, refresh the page to verify that the motion sensor thing is associate with the HCSR501MotionSensor device. Repeat the previous two steps for the RaspberryPiCameraRkgnExample and RaspberryPiScreen devices.

Create and deploy the flow configuration

1. Create the flow configuration.

Select the menu icon at the upper left of the page, and then select Flows to return to the Flows page. On the Flows page, select the box next to the flow that you just created, and then choose Create flow configuration.

2. Name the flow configuration.

On the Describe flow configuration page, select your flow, and then enter a flow configuration name. The flow configuration name can't contain spaces. Choose Greengrass, and then choose Next.

3. Configure the target.

On the Configure target page, enter the name of your Amazon S3 bucket and the AWS IoT Greengrass group to which your AWS IoT Greengrass core device belongs. Amazon S3 buckets are globally unique, so your bucket name will be different from the one in the following screen shot. Choose Next.
4. Select things for your deployment.

The Map Things page provides an interface for selecting the specific things that you'll include in your deployment. The menus under each device model in your deployment contain all of the things that you associated with the device model. Because you're getting started, the menus for each device model on this page will include only one thing (the thing that you've associated with each device model).

On the Map Things page, for motionSensor, select the motion sensor thing that you created earlier. Select the camera and screen things for the Camera and Screen device models. Choose Next.

5. View the trigger.

On the Define trigger page, the GraphQL that defines the motion event trigger for the flow appears in the editor. When the motion sensor detects motion, the ThingsGraph.startFlow function initiates the flow. You don't need to edit this code.

Choose Review.
6. Review and create.

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.

7. Deploy.

When the **Flow configuration created** message appears, choose **Deploy now**.

Refresh the **Deployments** page to verify that the flow has deployed. After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

**Run the flow**

Because the screen doesn't use information returned by the Amazon Rekognition service model, the steps for running this flow are the same as those in **Creating a flow with devices**.
Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you’ve created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Creating a flow with devices by using the AWS CLI

This topic contains the AWS CLI commands that create the flow in Creating a flow with devices. The setup instructions for this example are identical to the ones in that topic. Before starting with this example, follow all of the instructions in that topic up to Create and deploy the flow, and then return to this topic.

Install the CLI

To install the AWS CLI, follow the instructions in Installing the AWS CLI.

Create the flow by using the AWS CLI

The following steps describe how to create and deploy the flow by using the AWS CLI.

1. The following AWS IoT Things Graph Data Model (TDM) code contains the definition of the flow used in this example.

   Copy this code to a file. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID.

   ```json
   {  
     query SecurityFlow($camera: string!, $screen: string!) @workflowType(id:
       "urn:tdm:REGION/ACCOUNT ID/default:Workflow:SecurityFlow") {  
       variables {  
         cameraResult @property(id: "urn:tdm:aws/examples:property:CameraStateProperty") {  
       }  
       steps {  
         step(name: "Camera", outEvent: ["cameraStepDone"]) {  
           DeviceActivity(deviceModel: "urn:tdm:aws/examples:deviceModel:Camera", out:
             "cameraResult", deviceId: "${camera}" ) {  
             capture  
           }  
         }  
         step(name: "Screen", inEvent: ["cameraStepDone"] ) {  
           DeviceActivity(deviceModel: "urn:tdm:aws/examples:deviceModel:Screen", deviceId: "$\{screen\}" ) {  
             display(imageUrl: "$\{cameraResult.lastClickedImage\}" )  
           }  
         }  
       }  
     }  
   }
   ```

2. Enter the following command to create the flow. This command assumes that you’re working in a Linux or Unix environment. For other environments, use the equivalent of the cat utility.

   ```bash
   aws iotthingsgraph create-flow-template --definition
   language=GRAPHQL,text=""""$(cat PATH TO TDM FILE)"""
   ```
3. The following TDM code contains the definition of the system used in this example. The console creates this system for you.

Copy this code to a file. Replace the `REGION` and `ACCOUNT_ID` placeholders with your AWS Region and account ID.

```
{
  type SecurityFlow @systemType(id: "urn:tdm:REGION/ACCOUNT_ID/
    motionSensor: Device @thing(id: "urn:tdm:aws/examples:deviceModel:MotionSensor")
    screen: Device @thing(id: "urn:tdm:aws/examples:deviceModel:Screen")
    camera: Device @thing(id: "urn:tdm:aws/examples:deviceModel:Camera")
    SecurityFlow: Flow @workflow(id: "urn:tdm:REGION/ACCOUNT_ID/
  default:Workflow:SecurityFlow\")
  }
}
```

4. Enter the following command to create the system.

```
aws iotthingsgraph create-system-template --definition
language=GRAPHQL,text='''$(cat PATH TO TDM FILE)'''
```

5. Enter the following commands to associate the things you created in your registry with the device models used in the flow.

Replace the `MotionSensorName`, `CameraName`, and `ScreenName` placeholders with the names of the things you created earlier.

```
aws iotthingsgraph associate-entity-to-thing --thing-name "MotionSensorName" --entity-id "urn:tdm:aws/examples:Device:HCSR501MotionSensor"
aws iotthingsgraph associate-entity-to-thing --thing-name "CameraName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiCamera"
aws iotthingsgraph associate-entity-to-thing --thing-name "ScreenName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiScreen"
```

6. The following TDM code contains the definition of the flow configuration used in this example. The TDM definition is inside the `definition` object. For more information, see Creating flow configurations.

Copy this code to a file. Replace the `REGION` and `ACCOUNT_ID` placeholders with your AWS Region and account ID. Replace the `MotionSensorName`, `ScreenName`, and `CameraName` placeholders with the names of the things you created earlier.

```
{
  query Room215 @deployment(id: "urn:tdm:REGION/ACCOUNT_ID/default:Deployment:Room215\", systemId: "urn:tdm:REGION/ACCOUNT_ID/default:System:SecurityFlow\") {
    motionSensor(deviceId: "MotionSensorName")
    screen(deviceId: "ScreenName")
    camera(deviceId: "CameraName")
    triggers {MotionEventTrigger(description: \"a trigger\") {
      condition(expr: "devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent")
      action(expr: "ThingsGraph.startFlow('SecurityFlow', bindings[name == 'camera'].deviceId, bindings[name == 'screen'].deviceId)")
    }
  }
}
```
7. Enter the following command to create the flow configuration.

Replace the `GREENGRASS GROUP` and `S3 BUCKET` with the names of your AWS IoT Greengrass group and Amazon S3 bucket.

```bash
aws iotthingsgraph create-system-instance --definition language=GRAPHQL,text="$(cat PATH TO TDM FILE)" \\ --target GREENGRASS --greengrass-group-name GREENGRASS GROUP --s3-bucket-name S3 BUCKET
```

When the operation completes, the AWS CLI returns the following deployment summary as a JSON object. Use the `id` value in the `summary` block as the TDM URN of the flow configuration.

```
{
  "summary": {
    "status": "PENDING_DEPLOYMENT",
    "greengrassGroupName": "ThingsGraphGrnGr",
    "target": "GREENGRASS",
    "arn": "arn:aws:iotthingsgraph:REGION:ACCOUNT ID:default#Deployment#Room215",
    "updatedAt": 1555021747.176,
    "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Room215",
    "createdAt": 1555021747.176
  }
}
```

8. Enter the following command to deploy the flow configuration to your AWS IoT Greengrass group. Use the TDM URN value returned in the previous step as the value of the `id` parameter.

```bash
aws iotthingsgraph deploy-system-instance --id SYSTEM INSTANCE URN
```

9. To verify the deployment, follow the steps in either Run the Flow with Real Devices (p. 29) or Run the Flow with Mock Devices (p. 30).

Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you’ve created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Creating a flow with Lambda functions by using the AWS CLI

This topic contains the AWS CLI commands that create the flow in Create a flow with Lambda functions. The setup instructions for this example are identical to the ones in that topic. Before starting with this example, follow all of the instructions in that topic up to Create and deploy the flow, and then return to this topic.
AWS IoT Things Graph User Guide
AWS IoT Greengrass flow examples

Install the CLI

To install the AWS CLI, follow the instructions in Installing the AWS CLI.

Create the flow by using the AWS CLI

The following steps describe how to create and deploy the flow by using the AWS CLI.

1. The following AWS IoT Things Graph Data Model (TDM) code contains the definition of the flow used in this example.

Copy this code to a file. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID. Replace the S3 BUCKET IN SaveToS3 placeholder with the name of the Amazon S3 bucket that you're using in the SaveToS3 Lambda function.

```json
{
  query TextProcessing @workflowType(id: "urn:tdm:REGION/ACCOUNT ID/
default:Workflow:TextProcessing") {
    variables {
      getS3LambdaResult @property(id: "urn:tdm:aws/examples:property:getS3ObjectAsStringResponse")
      wordCountLambdaResult @property(id: "urn:tdm:aws/examples:property:wordCountResponse")
    }
    steps {
      step(name: "getS3Lambda", outEvent: ["getS3LambdaDone"]){
        WebserviceActivity(webservice: "urn:tdm:aws/examples:Service:getS3Lambda", out:
        "getS3LambdaResult") {
          getS3ObjectAsString(bucket: "S3 BUCKET IN SaveToS3", key: "HelloWorld.txt")
        }
      }
      step(name: "wordCountLambda", inEvent: ["getS3LambdaDone"], outEvent:
      ["wordCoundLambdaDone"]){
        WebserviceActivity(webservice: "urn:tdm:aws/examples:Service:wordCountLambda", out:
        "wordCountLambdaResult") {
          wordCount(message: "${getS3LambdaResult.message}\")
        }
      }
      step(name: "saveResponseLambda", inEvent: ["wordCoundLambdaDone"]){
        WebserviceActivity(webservice: "urn:tdm:aws/examples:Service:saveResponseLambda"
        ) {
          save(response: "${wordCountLambdaResult}\")
        }
      }
    }
  }
}
```

2. Enter the following command to create the flow. This command assumes that you're working in a Linux or Unix environment. For other environments, use the equivalent of the cat utility.

```
aws iotthingsgraph create-flow-template --definition
language=GRAPHQL,text=""""$(cat PATH TO TDM FILE)"""
```

3. The following TDM code contains the definition of the system used in this example. Because the flow contains no devices or device models, this system contains only the flow.
Copy this code to a file. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID.

```json
{
  type TextProcessing @systemType(id: "urn:tdm:REGION/ACCOUNT ID/
default:System:TextProcessing", description: "Text processing system") {
    TextProcessing: Flow @workflow(id: "urn:tdm:REGION/ACCOUNT ID/
default:Workflow:TextProcessing")
  }
}
```

4. Enter the following command to create the system.

```
aws iotthingsgraph create-system-template --definition
language=GRAPHQL,text='"$(cat PATH TO TDM FILE)"'
```

5. The following TDM code contains the definition of the flow configuration used in this example. The TDM definition is inside the `definition` object. For more information, see Creating flow configurations.

Copy this code to a file. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID.

```json
{
  query LambdaDeployment @deployment(id: "urn:tdm:REGION/ACCOUNT ID/
default:Deployment:LambdaDeployment", systemId: "urn:tdm:REGION/ACCOUNT ID/
default:System:TextProcessing") {
    triggers {
      TimeTrigger(description: "Time based trigger") {
        condition(expr: "every 60 seconds")
        action(expr: "ThingsGraph.startFlow('TextProcessing')")
      }
    }
  }
}
```

6. Enter the following command to create the flow configuration. Replace `GREENGRASS GROUP` and `S3 BUCKET` with the names of your AWS IoT Greengrass group and Amazon S3 bucket.

```
aws iotthingsgraph create-system-instance --definition
language=GRAPHQL,text='"$(cat PATH TO TDM FILE)"' 
--target GREENGRASS --greengrass-group-name GREENGRASS GROUP --s3-bucket-name S3 BUCKET
```

When the operation completes, the AWS CLI returns the following deployment summary as a JSON object. Use the `id` value in the `summary` block as the TDM URN of the flow configuration.

```json
{
  "summary": {
    "status": "PENDING_DEPLOYMENT",
    "greengrassGroupName": "ThingsGraphGrnGr",
  }
}
```
7. Enter the following command to deploy the flow configuration to your AWS IoT Greengrass group. Use the TDM URN value returned in the previous step as the value of the id parameter.

```
aws iotthingsgraph deploy-system-instance --id SYSTEM INSTANCE URN
```

Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you've created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Creating a flow with devices and a service by using the CLI

This topic contains the AWS CLI commands that create the flow in Creating a flow with devices and a service. The setup instructions for this example are identical to the ones in that topic. Before starting with this example, follow all of the instructions in that topic up to Create and deploy the flow, and then return to this topic.

Install the CLI

To install the AWS CLI, follow the instructions in Installing the AWS CLI.

Create the flow by using the AWS CLI

The following steps describe how to create and deploy the flow by using the AWS CLI

1. The following AWS IoT Things Graph Data Model (TDM) code contains the definition of the flow used in this example. Copy this code to a file. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID.

```json
{
    "target": "GREENGRASS",
    "arn": "arn:aws:iotthinggraph:REGION:ACCOUNT ID:default#Deployment#LambdaDeployment",
    "updatedAt": 1555022420.184,
    "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:LambdaDeployment",
    "createdAt": 1555022420.184
}
}
```

```json
query RekognitionFlow($cameraRkgnExample: string!, $screen: string!) @workflowType(id: \"urn:tdm:REGION/ACCOUNT ID/default:Workflow:RekognitionFlow\") {
    variables {
        cameraRkgnExampleResult @property(id: \"urn:tdm:aws/examples:property:CameraStatePropertyRkgnExample\")
        rekognitionResult @property(id: \"urn:tdm:aws:Property:Json\")
    }
    steps {
        step(name: \"CameraRkgnExample\", outEvent: [\"cameraStepDone\"]) {
            DeviceActivity(deviceModel: \"urn:tdm:aws/examples:deviceModel:CameraRkgnExample\", out: \"cameraRkgnExampleResult\", deviceId: \"${cameraRkgnExample}\")
            capture
        }
        step(name: \"Rekognition\", inEvent: [\"cameraStepDone\"], outEvent: [\"rekognitionStepDone\"])
```
2. Enter the following command to create the flow. This command assumes that you're working in a Linux or Unix environment. For other environments, use the equivalent of the `cat` utility.

```bash
aws iotthingsgraph create-flow-template --definition language=GRAPHQL,text="""$(cat PATH TO TDM FILE)"""
```

3. The following TDM code contains the definition of the system used in this example. The console creates this system for you.

Copy this code to a file. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID.

```json
{
  type RekognitionFlow @systemType(id: "urn:tdm:REGION/ACCOUNT ID/default:System:RekognitionFlow") {
    motionSensor: Device @thing(id: "urn:tdm:aws/examples:deviceModel:MotionSensor")
    cameraRkgnExample: Device @thing(id: "urn:tdm:aws/examples:deviceModel:CameraRkgnExample")
    screen: Device @thing(id: "urn:tdm:aws/examples:deviceModel:Screen")
    RekognitionFlow: Flow @workflow(id: "urn:tdm:REGION/ACCOUNT ID/default:Workflow:RekognitionFlow")
  }
}
```

4. Enter the following command to create the system.

```bash
aws iotthingsgraph create-system-template --definition language=GRAPHQL,text="""$(cat PATH TO TDM FILE)"""
```

5. Enter the following commands to associate the things you created in your registry with the device models used in the flow.

Replace the `MotionSensorName`, `CameraName`, and `ScreenName` placeholders with the names of the things you created earlier.

```bash
aws iotthingsgraph associate-entity-to-thing --thing-name "MotionSensorName" --entity-id "urn:tdm:aws/examples:Device:HCSR501MotionSensor"
```
aws iotthingsgraph associate-entity-to-thing --thing-name "CameraName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiCameraRkgnExample"
aws iotthingsgraph associate-entity-to-thing --thing-name "ScreenName" --entity-id "urn:tdm:aws/examples:Device:RaspberryPiScreen"

6. The following TDM code contains the definition of the flow configuration used in this example. The TDM definition is inside the definition object. For more information, see Creating flow configurations.

Copy this code to a file. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID. Replace the MotionSensorName, ScreenName, and CameraName placeholders with the names of the things you created earlier.

```json
{
    query Lobby @deployment(id: "$urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby"),
    systemId: "$urn:tdm:REGION/ACCOUNT ID/default:System:RekognitionFlow"
    {
        motionSensor(deviceId: "$MotionSensorName")
        screen(deviceId: "$ScreenName")
        cameraRkgnExample(deviceId: "$CameraName")
        triggers {
            MotionEventTrigger(description: "a trigger") {
                condition(expr: "devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent")
                action(expr: "ThingsGraph.startFlow('RekognitionFlow', bindings[name == 'cameraRkgnExample'].deviceId, bindings[name == 'screen'].deviceId)"
            }
        }
    }
}
```

7. Enter the following command to create the flow configuration. Replace the GREengrass GROUP and S3 BUCKET with the names of your AWS IoT Greengrass group and Amazon S3 bucket.

```bash
aws iotthingsgraph create-system-instance --definition language=GRAPHQL,text=""$(cat PATH TO TDM FILE)"" 
--target GREengrass --greengrass-group-name GREengrass GROUP --s3-bucket-name S3 BUCKET
```

When the operation completes, the AWS CLI returns the following deployment summary as a JSON object. Use the id value in the summary block as the TDM URN of the flow configuration.

```json
{
    "summary": {
        "status": "PENDING_DEPLOYMENT",
        "greengrassGroupName": "ThingsGraphGrnGr",
        "target": "GREengrass",
        "arn": "arn:aws:iotthingsgraph:REGION:ACCOUNT ID:default#Deployment#Lobby",
        "updatedAt": 1555022871.418,
        "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby",
        "createdAt": 1555022871.418
    }
}
```

8. Enter the following command to deploy the flow configuration to your AWS IoT Greengrass group. Use the TDM URN value returned in the previous step as the value of the id parameter.
Delete the flow and flow configuration (optional)

For instructions on how to undeploy a flow configuration, and delete the flow configuration and flow that you've created, see Deleting flow configurations and Deleting systems, flows, and namespaces in Lifecycle management for AWS IoT Things Graph entities, flows, systems, and deployments.

Using the example mock devices

Two of the examples in this Getting Started section provide ready-to-use Python scripts to mimic the behavior of real MQTT devices in a flow: Creating a flow in the cloud with devices and Creating a flow in an AWS IoT Greengrass group with devices. These scripts subscribe and publish to MQTT topics that AWS IoT Things Graph uses to mediate the steps in a flow.

This topic explains what these scripts do, and how you can modify them to test your flows when you don't have real devices.

You can find the scripts here:

- **CloudMockDevices.zip** – Contains the cloud mock device scripts.
- **MockDevices.zip** – Contains the AWS IoT Greengrass mock device scripts.

What the mock devices do

In flows that include MQTT devices, AWS IoT Things Graph uses MQTT topics to mediate the communications between the devices in a flow.

**Motion Sensor**

In the example flows that include a motion sensor, a camera, and a screen, the motion sensor publishes a JSON object that indicates that it detects motion. Because the motion sensor mock device isn't real, it sends the message every 10 seconds.

**Camera**

The camera must both subscribe and publish to MQTT topics because it takes input and generates output. The camera subscribes to a topic where AWS IoT Things Graph publishes the instruction to take a picture. After the camera takes the picture, it publishes a JSON object containing the URI of the resulting image to another topic.

**Screen**

The screen subscribes to a topic where AWS IoT Things Graph publishes the image URI. Because the screen mock device isn't real, it simply prints the image URI to standard output.

**Flow Sequence**

You could use the flow's components in a more real-world scenario, such as a door entry system in which a facial recognition service verifies the identity of the person who triggers the motion sensor. A door opens or remains closed depending on the result that the service sends.
The following graphic shows how the example flow works.

The motion sensor publishes to the /motion topic. The camera subscribes to the /capture topic and publishes to the /capture/finished topic. The screen subscribes to the /display topic.

Each MQTT topic used in the flow begins with the name of the thing associated with a device in the flow. The following list contains the names of each MQTT topic and the device that subscribes and/or publishes to it. This is the sequence that the devices follow in the flow:

1. The motion sensor publishes to Motion Sensor Thing Name/motion.
2. The camera subscribes to Camera Thing Name/capture.
3. The camera publishes to Camera Thing Name/capture/finished.
4. The screen subscribes to Screen Thing Name/display.

Now let's look at the details of what each mock device does in this sequence.

**Motion sensor**

The motion sensor mock device sends the following JSON to Motion Sensor Thing Name/motion every 10 seconds.

```json
{"isMotionDetected": true}
```

When the motion sensor publishes this message, it tells AWS IoT Things Graph that the motion sensor's state changed. The isMotionDetected value and Boolean data type match the single field in the MotionSensorState entity, as defined in the Aukru HCSR501 motion sensor definition.

```graphql
type MotionSensorState @stateType(id: "urn:tdm:aws/examples:State:MotionSensorState") {
  isMotionDetected: Boolean @property(id: "urn:tdm:aws:property:Boolean")
}
```

The capability implementation in the Aukru HCSR501 motion sensor definition specifies the MQTT topic to which the motion sensor publishes the message.

**Note**

If the MQTT message occurs as part of a device event, you'll find the relevant topic or topics in the Event block. If it occurs as part of a device action, you'll find the relevant topics in the Action block.

```graphql
MQTT {
  MotionSensorCapability(id: "urn:tdm:aws/examples:capability:MotionSensorCapability") {
    ... 
    Event(name: "StateChanged") {
      Subscribe(topic: "${systemRuntime.deviceId}/motion") {
        responsepayload(property: "urn:tdm:aws/examples:property:MotionSensorStateProperty")
      }
    }
  }
}
```
When the motion sensor's state changes, AWS IoT Things Graph receives the message on the topic specified in the `StateChanged` event: the device ID (the name of the thing in the AWS IoT registry that is being used in the flow) and `/motion`. This is the topic to which the motion sensor publishes.

**Camera**

The camera mock device listens for capture commands that AWS IoT Things Graph publishes to `CameraThing Name/capture`. Because the flow doesn't specify any input values for the camera action, the JSON object published to this topic is empty. The camera then publishes the following JSON to `CameraThing Name/capture/finished`:

```
{"imageUri":"Image URI"}
```

The `imageUri` value and string data type match the single field in the `CameraState` entity, as defined in the Raspberry Pi camera definition.

```typescript
type CameraState @stateType(id: "urn:tdm:aws/examples:state:CameraState") {
  lastClickedImage : Uri @property(id: "urn:tdm:aws:property:String")
}
```

The capability implementation in the Raspberry Pi camera definition specifies the MQTT topics to which the camera subscribes and publishes messages. These topics are inside the Publish block of the Action block.

```typescript
MQTT {
  CameraCapability(id: "urn:tdm:aws/examples:capability:CameraCapability") {
    ...
    Action(name: "capture") {
      Publish {
        Request(topic: "$macro(${systemRuntime.deviceId}/capture") { 
          params
        }
        Response(topic: "$macro(${systemRuntime.deviceId}/capture/finished") { 
          responsePayload(property: "urn:tdm:aws/examples:property:CameraStateProperty")
        }
      }
    }
  }
}
```

When the value of the property of `lastClickedImage` in the device's state changes, AWS IoT Things Graph publishes an empty request to the `device ID (thing name)/capture` topic. It then expects the camera to publish a response to the `device ID (thing name)/capture/finished` topic.

**Screen**

After receiving the `imageUri`, AWS IoT Things Graph publishes a JSON object that is identical to the one that the camera sends to the `Screen Thing Name/display` topic.
Adapting the mock devices

The capability implementation in the Raspberry Pi screen definition specifies the MQTT topic to which the screen subscribes. This topic is inside the Publish block of the Action block.

```json
MQTT {
  ScreenCapability(id: "urn:tdm:aws/examples:capability:ScreenCapability") {
    ...
    Action(name: "display") {
      params {
        param(name: "imageUrl" property: "urn:tdm:aws:property:String")
      }
    }
    Publish {
      Request(topic: 
        "$macro(${systemRuntime.deviceId}/display)"
      ) {
        params {
          param(name: "imageUri", property: "urn:tdm:aws/examples:property:CameraStateProperty", value: 
            "${imageUrl.value}"
        )
      }
    }
  }
}
```

AWS IoT Things Graph gets the image URI from the camera, and then passes it as a parameter to the Action implementation. The Action implementation tells AWS IoT Things Graph to publish the image URI to the device ID (thing name)/display topic.

Adapting the mock devices

You can write your own mock devices using your preferred languages and SDKs.

The Python mock devices show the following:

- How to use the AWS IoT Device SDK for Python and the AWS IoT Greengrass Core SDK to subscribe and post to MQTT topics in the cloud, and in an AWS IoT Greengrass group.
- At a high level, how MQTT devices interact with AWS IoT Things Graph when they perform steps in a flow.
- The mock device scripts in Creating a flow in an AWS IoT Greengrass group with devices also demonstrate how to discover and connect to an AWS IoT Greengrass group so that devices in the group can interact with each other. This is the necessary first step before devices start sending messages to AWS IoT Things Graph when it's running on an AWS IoT Greengrass core device.

When you adapt the mock devices, you need to change only the MQTT topics and JSON payloads to match the topics, inputs, and outputs that you've specified in your own flows.

When you adapt the Python mock device scripts, look for the lines containing `myAWSIoTMQTTClient.subscribe` and `myAWSIoTMQTTClient.publish`. For example, the `cloudcamera.py` script subscribes to the thingName/capture topic on line 129 of the file.

```python
myAWSIoTMQTTClient.subscribe(thingName + "/capture", 0, customCallback)
```

The first parameter of the `subscribe` method specifies the topic to which the mock device subscribes. You can use this method to subscribe to any MQTT topic that you've specified in your flow. If you want the mock device to do something other than print a message whenever it receives a message on the topic, you can create your own callback function.
The `cloudcamera.py` script publishes to the `thingName/capture/finished` topic on line 120 of the file.

```python
myAWSIoTMQTTClient.publish(thingName + "/capture/finished", messageJson, 0)
```

The first parameter of the `publish` method specifies the topic to which the mock device publishes. You can use this method to publish to any MQTT topic that you've specified in your flow. The second parameter specifies the JSON object that contains the message to publish. You can change this value to match the input and output values that you've specified in your flow.
Modeling entities

This section describes how to create and work with the entities that compose a workflow (flow) after you create them in the AWS IoT Things Graph Data Model (TDM).

Entities include Properties, States, Events, Actions, Capabilities, Mappings, Devices, and Services.

After you create these entities in TDM, you upload them to your namespace by using the UploadEntityDefinitions API.

For complete example device models and services, see Example device and service definitions.

Topics

- Modeling by device communication protocol (p. 57)
- MQTT device modeling 101 (p. 62)
- Modbus device modeling 101 (p. 68)
- Service modeling 101 (p. 73)
- Creating and uploading entities (p. 80)
- Versioning and entity modeling (p. 88)
- Viewing models and things in the AWS IoT Things Graph console (p. 88)

Modeling by device communication protocol

AWS IoT Things Graph supports the MQTT and Modbus protocols for device communication in a flow. The protocol that the device uses affects the way you define the device.

The topics in this section describe how you define devices and their interactions.

AWS IoT Things Graph supports the following protocols:

- MQTT
- Modbus

For information about defining services that use HTTP (REST) and AWS Lambda functions, see Service in the AWS IoT Things Graph Data Model Reference.

MQTT

AWS IoT Things Graph enables interaction through the MQTT protocol by specifying MQTT as the communication protocol in the device definition. This topic describes how to model an MQTT device in GraphQL.

The following GraphQL shows how to define a device (a camera) that uses the MQTT protocol. This example assumes that the state, event, and capability implemented in the device are already defined.
For more information about defining devices, see the Things data model reference, and the Device construct specifically.

```graphql
{
  type DemoCamera @deviceModel(id: "urn:tdm:REGION/ACCOUNT ID/
default:deviceModel:DemoCamera",
capability: "urn:tdm:REGION/ACCOUNT ID/default:capability:DemoCameraCap")
  {ignore:void}
    query DemoPhyCamera @device(id: "urn:tdm:REGION/ACCOUNT ID/
default:device:DemoPhyCamera",
deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DemoCamera") {
      MQTT {
        DemoCameraCap(id: "urn:tdm:REGION/ACCOUNT ID/
default:capability:DemoCameraCap") {
          State {
            lastClickedImage(name: "lastImage", property: "urn:tdm:aws:property:String")
          }
          Action(name: "capture") {
            Publish {
              Request(topic: "${systemRuntime.deviceId}/capture") {
                params
              }
              Response(topic: "${systemRuntime.deviceId}/capture/finished") {
                responsePayload(property: "urn:tdm:REGION/ACCOUNT ID/
default:property:DemoCameraStateProperty")
              }
            }
          }
        }
      }
    }
}
```

Key elements:

- **Implementations of the device's State and Action** - A device implementation can also implement a device Event.
- **The Publish block inside the Action implementation** - This block contains the Request and Response definitions.
- **The Request definition** - This definition specifies the MQTT request topic and the parameters, if any, that are sent to it.
- **The Response definition** - This optional definition specifies the MQTT response topic (if one exists) and the payload that is sent to it.

The following GraphQL defines another GraphQL device (a motion sensor). This example implements a device Event that contains a Subscribe block. This block specifies the MQTT topic to which the motion sensor subscribes.

```graphql
{
  type DemoMotionSensor @deviceModel(id: "urn:tdm:REGION/ACCOUNT ID/
default:deviceModel:DemoMotionSensor",
capability: "urn:tdm:REGION/ACCOUNT ID/default:capability:DemoMotionSensorCap")
  {ignore:void}
    query DemoPhyMotionSensor @device(id: "urn:tdm:REGION/ACCOUNT ID/
default:device:DemoPhyMotionSensor", deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DemoMotionSensor") {
      MQTT {
        DemoMotionSensorCap(id: "urn:tdm:REGION/ACCOUNT ID/
default:capability:DemoMotionSensorCap") {
          State {
            lastClickedImage(name: "lastImage", property: "urn:tdm:aws:property:String")
          }
          Action(name: "motion") {
            Subscribe {
              topic: "system/motion"
              data: true
            }
          }
        }
      }
    }
```
Modbus

AWS IoT Things Graph enables interaction through the Modbus protocol by specifying Modbus as the communication protocol in the device definition.

This topic describes how to model a Modbus device in GraphQL. It also describes the types of Modbus interactions that the Things Data Model (TDM) currently supports, and the data transformations that occur when data types are modeled using the AWS IoT Greengrass Modbus-RTU Protocol Adapter.

Prerequisites

- Modbus devices registered with credentials in the AWS IoT things registry.

For instructions on how to create things in the registry, see Register a device in the registry.

**Note**

To add your devices to your AWS IoT Greengrass group, you need to register them as things with credentials, even though your devices don't need to use these credentials when your flows are running.

GraphQL for defining Modbus devices

The following GraphQL shows how to define a device (a sprinkler) that uses the Modbus protocol. This example assumes that the states, actions, and capability implemented in the device are already defined. For more information about defining devices, see the Things data model reference, and the **Device** construct specifically.

```graphql
{
  type Sprinkler @deviceModel(id: "urn:tdm:REGION/ACCOUNT ID/
default:DeviceModel:Sprinkler",
capability: "urn:tdm:REGION/ACCOUNT ID/
default:Capability:SprinklerCapability) {
    ignore:void
  }
}
```
query ModbusSprinkler @device(id: "urn:tdm:REGION/ACCOUNT ID/
    default:Device:MyModbusSprinkler",
    deviceModel: "urn:tdm:REGION/ACCOUNT ID/
    default:DeviceModel:Sprinkler") {
  Modbus(serverId:"2") {
    SprinklerCapability (id: "urn:tdm:REGION/ACCOUNT ID/
        default:Capability:SprinklerCapability") {
      State {
        valveState(name: "valveState", property:"urn:tdm:aws:Property:Boolean")
        headAngle(name: "headAngle", property:"urn:tdm:REGION/ACCOUNT ID/
            default:Property:Angle")
      }
      Action(name: "actuateSprinklerHead") {
        Params {
          param(name:shouldActivate, property:
            "urn:tdm:aws:Property:Boolean")
        }
        WriteSingleCoil {
          Request(Address: 1) {
            params {
              param(name:shouldActivate,
                  property: "urn:tdm:aws:Property:Boolean",
                  value: "${shouldActivate}"简洁的表达)
            }
          }
        }
      }
      Action(name: "getSprinklerHeadAngle") {
        Params
        ReadHoldingRegisters {
          Response {
            responsePayload(property: "urn:tdm:aws:Property:UInt16")
          }
        }
      }
      Action(name: "setSprinklerHeadAngle") {
        Params {
          param(name:angle, property: "urn:tdm:aws:Property:UInt16")
        } 
        WriteSingleRegister {
          Request(Address: 1) {
            params {
              param(name:angle,
                  property: "urn:tdm:aws:Property:UInt16",
                  value: "${angle}"简洁的表达)
            }
          }
        }
      }
    }
  }
}

Key elements:

- The `serverId` argument passed to the Modbus protocol block - The value of this argument specifies the Modbus endpoint ID to which AWS IoT Things Graph sends messages.
- Implementations of the device’s State and Actions - Modbus defines no event protocol for messages to be sent to the master, so the device definition contains no Event implementations.
• **Modbus request and response operations** - Operations such as `WriteSingleCoil`, `ReadHoldingRegisters`, and `WriteSingleRegister` map precisely to the operations in the AWS IoT Greengrass Modbus-RTU Protocol Adapter.

### Interaction flow

AWS IoT Things Graph interacts with Modbus devices by using the AWS IoT Greengrass Modbus RTU Protocol Adapter connector. The following diagram shows the flow that occurs when a flow invokes a Modbus device.

![Flow Diagram](image)

### Payload translations

When AWS IoT Things Graph interacts with Modbus devices, most simple types and lists of simple types that are supported in the AWS IoT Things Graph data model (TDM) can be mapped to Modbus payloads. However, complex objects can’t be mapped.

The following tables describe the expected behavior when AWS IoT Things Graph serializes and deserializes Modbus data.

#### Coil interactions

<table>
<thead>
<tr>
<th>TDM data type</th>
<th>Modbus serialization</th>
<th>Example TDM data</th>
<th>Example Modbus registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Boolean values (a single Boolean value is treated as a list of length 1)</td>
<td>0 if the value is false, 1 if value is true. Written to contiguous coils.</td>
<td>0 [0, 1, 1, 0]</td>
<td>[0, 1, 1, 0]</td>
</tr>
<tr>
<td>Single integer value (including all Int and UInt values)</td>
<td>0 if the number == 0; otherwise, 1. Written to contiguous coils.</td>
<td>(UInt16)13 (Int64)0</td>
<td>1 0</td>
</tr>
</tbody>
</table>
### MQTT device modeling 101

This topic describes what you need to think about and plan for when you're creating models for your MQTT devices.

We'll use a motion sensor and a camera as reference points, but you can apply the process that this topic describes to any device that uses the MQTT communication protocol. The devices that we model here are the Aukru HC-SR501 motion sensor and the Raspberry Pi camera.

<table>
<thead>
<tr>
<th>TDM data type</th>
<th>Modbus serialization</th>
<th>Example TDM data</th>
<th>Example Modbus registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean or list of Boolean values</td>
<td>Writes <strong>0x01</strong> to each register where Boolean is true, and <strong>0x00</strong> where Boolean is false.</td>
<td>true</td>
<td>[0x0001]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[true, true, false]</td>
<td>[0x0001, 0x0001, 0x0000]</td>
</tr>
<tr>
<td>Int8 and UInt8 values</td>
<td>Writes integers, two per register. If there is an odd number of integers (including one integer), the value is written to MSB (most significant byte).</td>
<td>(Int8)1</td>
<td>[0x1000]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([Int8]3, (Int8)5, (Int8)4)</td>
<td>[0x0305][0x04000]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([UInt8]14, (UInt8)7)</td>
<td>[0xD700]</td>
</tr>
<tr>
<td>Int16 and UInt16 values</td>
<td>Writes integers, one per register.</td>
<td>([Int16]17, (Int16)1)</td>
<td>[0x0010][0x0001]</td>
</tr>
<tr>
<td>Int32 and UInt32 values</td>
<td>Writes one integer per two registers. MSB is written first.</td>
<td>[0x00123400, 0x00F0]</td>
<td>[0x0012][0x3400][0x0000][0x00F0]</td>
</tr>
<tr>
<td>Int64 and UInt64 values</td>
<td>Writes one integer per four registers. MSB is written first.</td>
<td>0x11223344</td>
<td>[0x0000][0x0000][0x1122][0x3344]</td>
</tr>
<tr>
<td>Float32 values</td>
<td>Writes one float per two registers.</td>
<td>[3.14159, 2.71828]</td>
<td>[0x4049][0x0FD0][0x402D][0xF84D]</td>
</tr>
<tr>
<td>Float64 values</td>
<td>Writes one float per four registers.</td>
<td>3.14159</td>
<td>[0x4009][0x21F9][0xF01B][0x866E]</td>
</tr>
<tr>
<td>String</td>
<td>Writes the string as UTF-8 encoded text. If the string is an odd number of bytes, the last byte is written to the MSB of the last register. A list of strings isn't allowed.</td>
<td>&quot;hello&quot;</td>
<td>[0x6865][0x6C6C][0x6F00]</td>
</tr>
</tbody>
</table>
Creating your abstract device model

The Aukru-HC-SR501.zip and RaspberryPiCamera.zip files contain all of the GraphQL code discussed in this topic.

When you define a device, you need to create two pieces. The first piece is the abstract device model. This piece generically defines what a type of device does. The second piece is the device definition, which implements the model. The device definition specifies the communication protocol used by the device. Devices that use different protocols can inherit from the same device model.

Before you define your device, look in the AWS IoT Things Graph console to determine whether an abstract model for the type of device you're defining already exists. If so, you can skip to the task of defining your device.

If you can't find an existing abstract device model for your device in the console, you have to define one. For this exercise, assume that no model for a motion sensor device currently exists.

Creating your abstract device model

The first piece of your device definition is its abstract representation in a device model. The device model contains the capability. You'll define the capability later, but now you can assign its AWS IoT Things Graph Data Model (TDM) URN in the device model. The following GraphQL contains the motion sensor and camera device models. (The REGION and ACCOUNT ID values are specific to your account.)

Motion Sensor

```graphql
type MotionSensor @deviceModel(id: "urn:tdm:aws/examples:deviceModel:MotionSensor")
```
Creating your device's capability (motion sensor)

The motion sensor device capability represents what happens to the sensor when an event occurs and how the sensor responds to this event.

The motion sensor device capability requires the following two TDM entities:

- A State that represents whether the sensor has detected motion
- An Event that represents the notification that the motion sensor sends when it detects motion

The Event also requires a complex Property that represents an instance of the device's State. This Property is the payload that the device publishes to an MQTT topic whenever it detects motion.

Start defining your device's capability by thinking about the functions that it serves. A motion sensor device detects when a body or part of a body moves within the sensor's range. When a motion sensor detects a body in motion, it generates a notification that this event has occurred. Therefore, its capability definition requires only a single event.

A motion detector also has two states: no motion detected and motion detected. The motion detected event changes the state, so you need to include the state and the event in your capability. You can represent this state as a Boolean value. The following GraphQL defines this state.

```graphql
type MotionSensorState @stateType(id: "urn:tdm:aws/examples:State:MotionSensorState") {
  isMotionDetected: Boolean @property(id: "urn:tdm:aws:property:Boolean")
}
```

When it detects motion, the sensor sends a message to an MQTT topic. This message contains a property that describes the current state of the sensor (in this case, the state of its isMotionDetected property changing from false to true). To make this possible, you need to create a complex property that is an instance of the motion sensor's state. The following GraphQL defines this property.

```graphql
type MotionSensorStateProperty @propertyType(id: "urn:tdm:aws/examples:property:MotionSensorStateProperty",
  instanceOf: "urn:tdm:aws/examples:State:MotionSensorState",
  description: "Property representing the motion sensor state") {ignore:void}
```

Use this property as the payload that the motion sensor device sends when the motion detected event triggers a change in its state. The following GraphQL defines this Event and specifies the complex property as the payload that it publishes to an MQTT topic.
Creating your device's capability (camera)

The camera device capability represents the action of a camera capturing an image. The camera device capability requires the following two TDM entities:

- A State that represents the last image taken by the camera
- An Action that represents the notification that the camera sends when it captures an image

The Action also requires a complex Property that represents an instance of the device's State. This Property is the payload that the device publishes to an MQTT topic whenever it captures an image.

A camera captures and stores images. We'll store the last image that the camera has captured in its state. The following GraphQL defines this state.

```graphql
type CameraState @stateType(id: "urn:tdm:aws/examples:state:CameraState") {
  lastClickedImage : Uri @property(id: "urn:tdm:aws:property:String")
}
```

After it captures an image, the camera sends a message to an MQTT topic. This message contains a property that describes the current state of the camera (in this case, the location of the last image it has captured). Just as you did with the motion sensor, you need to create a complex property that is an instance of the camera's state. The following GraphQL defines this property.

```graphql
type MotionSensorEvent @eventType(id: "urn:tdm:aws/examples:event:MotionSensorEvent", payload: "urn:tdm:aws/examples:property:MotionSensorStateProperty") {ignore: void}
```
Use this property as the payload that the camera sends when it captures an image and its state changes. The following GraphQL defines this action and specifies the complex property as the value that it publishes to an MQTT topic.

```graphql
type Capture @actionType(id: "urn:tdm:aws/examples:action:Capture") {
  return: CameraStateProperty @property(id: "urn:tdm:aws/examples:property:CameraStateProperty")
}
```

You specify the MQTT topic to which the device publishes this payload later when you create your device definition.

Now you're ready to define the device's capability. The capability contains both the CameraState that you created earlier and an instance of the Capture action, which is conveniently named capture. The TDM URN that uniquely identifies this capability matches the one that you used when you created the camera abstract device model.

```graphql
type CameraCapability @capabilityType(id: "urn:tdm:aws/examples:capability:CameraCapability") {
  STATE: CameraState @state(id: "urn:tdm:aws/examples:state:CameraState")
  capture: Capture @action(id: "urn:tdm:aws/examples:action:Capture")
}
```

You use the capture event name when you implement the device's capability in the device definition.

## Creating your device definitions

Now that you have an abstract device model that contains a fully defined capability, you can write the definition for your specific device. The abstract device model and capability specify what a motion sensor does at a generic level. The device definition specifies how the specific device you want to use in a flow deployment interacts with AWS IoT Things Graph.

Your device definition contains an MQTT block. This block implements the capability that you defined earlier. It also specifies the MQTT topic to which the device publishes its payload (the complex property that specifies that the device's state has changed). The following GraphQL contains the motion sensor device definition.

```graphql
query HCSR501MotionSensor @device(id: "urn:tdm:aws/examples:device:HCSR501MotionSensor", deviceModel: "urn:tdm:aws/examples:deviceModel:MotionSensor") {
  MQTT {
    MotionSensorCapability(id: "urn:tdm:aws/examples:capability:MotionSensorCapability") {
      state {
        isMotionDetected(name: "isMotionDetected", property: "urn:tdm:aws:property:Boolean")
      }
      Event(name: "StateChanged") {
        Subscribe(topic: "#macro(${systemRuntime.deviceId}/motion")
```
Creating your device definitions

```graphql
responsePayload(property: "urn:tdm:aws/examples:property:MotionSensorStateProperty")
}
}
}
```

Key features:

- The two parameters included after the `@device` declaration are the IDs of the device and the abstract device model from which the device inherits.
- The MQTT block contains the state that you created earlier, and the Boolean property that you specified in the state definition.
- The MQTT block also contains the Event block. This block takes a name argument whose value matches the event name (StateChanged) that you specified in the capability definition.
- An event uses a Subscribe block to specify where the event notification will be published. The Subscribe block takes a topic argument that specifies the MQTT topic to which the device publishes its notification. In this case, the definition uses the macro keyword to include the value of the system variable that stores the name of the thing (in the AWS IoT registry) that you've associated with the device in a specific flow deployment. This is a good way to ensure that things in multiple flow deployments are publishing to unique MQTT topics.
- The responsePayload block takes a property argument that specifies the payload that the device sends to the MQTT topic. In this case, the device sends the value of the complex property that you created earlier.

The following GraphQL contains the camera device definition.

```graphql
query RaspberryPiCamera @device(id: "urn:tdm:aws/examples:device:RaspberryPiCamera", deviceModel: "urn:tdm:aws/examples:deviceModel:Camera") {
MQTT {
  CameraCapability(id: "urn:tdm:aws/examples:capability:CameraCapability") {
    state {
      lastClickedImage(name: "lastImage", property: "urn:tdm:aws:property:String")
    }
    Action(name: "capture") {
      Publish {
        Request(topic: "macro(${systemRuntime.deviceId}/capture") {
          params
        }
        Response(topic: "macro(${systemRuntime.deviceId}/capture/finished)") {
          responsePayload(property: "urn:tdm:aws/examples:property:CameraStateProperty")
        }
      }
    }
  }
}
```

Key features:

- The two parameters included after the `@device` declaration are the IDs of the device and the abstract device model from which the device inherits.
• The MQTT block contains the state that you created earlier, and the String property that you specified in the state definition.

• The MQTT block also contains the Action block. This block takes a name argument whose value matches the event name (StateChanged) that you specified in the capability definition.

• An action uses a Publish block to specify where the action publishes its output. The Publish block contains Request and Response blocks. The Request block specifies the MQTT topic that sets the action in motion when notifications are published to it. You use the Response block to send a notification that indicates that the action is complete. This notification contains the location of the last image that the camera has captured.

• The responsepayload block inside the Response block takes a property argument that specifies the payload that the device sends to the MQTT topic. In this case, the device sends the value of the complex property that you created earlier.

The Aukru-HC-SR501.zip and RaspberryPiCamera.zip files contain all of the GraphQL code discussed in this topic. Download them to work with the code, and then upload any of your changes to AWS IoT Things Graph yourself.

**Modbus device modeling 101**

This topic describes what you need to think about and plan for when you're creating models for your Modbus devices.

We'll use a weather monitoring device as a reference point, but you can apply the process that this topic describes to any device that uses the Modbus communication protocol. The weather monitoring device that we model here is the Elsner P03 Weather Station (be sure to read the specification's description).

The Elsner-Weather-Station.zip file contains all of the GraphQL code discussed in this topic.

The Elsner P03 Weather Station measures conditions in the physical environment. From the device's description in the specification, we know that this device performs the following functions:

- Measures brightness, with sensors for east, south, and west
- Recognizes twilight and dawn
- Measures wind strength
- Measures temperature
- Recognizes the presence of precipitation

When you define a device, you need to create two pieces. The first piece is the abstract device model. This piece generically defines what a type of device does. The second piece is the device definition, which implements the model. The device definition specifies the communication protocol used by the device. Devices that use different protocols can inherit from the same device model.

Before you define your device, look in the AWS IoT Things Graph console to determine whether an abstract model for the type of device you're defining already exists. If so, you can skip to the task of defining your device.
Creating your abstract device model

Begin by thinking about the function or functions that your device serves. A weather monitoring device, or weather station, measures various conditions in the physical environment, such as temperature, wind speed, and the brightness of sunlight. In the AWS IoT Things Graph Data Model (TDM), this collection of actions that the device can perform is its capability.

Because this device uses the Modbus protocol, AWS IoT Things Graph communicates only with the master polling device. The master device doesn't receive inputs from other devices, so when you model a Modbus device in TDM, you don't need to define any events. Modbus device capabilities contain only actions.

Knowing this, you can start by creating an abstract TDM device model. (The REGION and ACCOUNT ID values are specific to your account. The Suffix value is 1 until you create different versions of your weather station.)

```yaml
type WeatherStation @deviceModel(id: "urn:tdm:{REGION}/{ACCOUNT_ID}/default:deviceModel:WeatherStation_{SUFFIX}",
capability: "urn:tdm:{REGION}/{ACCOUNT_ID}/default:capability:WeatherStationCapability_{SUFFIX}",{}
IGNORE: VOID
```
Creating your abstract device model

Next, you need to define the capability that you included in your abstract model. Look through your product specification and note all of the actions that the master device performs and the registers that it uses to read and write data. A typical weather station device needs to read data only from the devices that it’s polling, so all of your actions will be “get” operations.

Because you’re creating a model for a weather station, you need to create a capability that includes recognizing rain and measuring temperature, wind speed, and the brightness of sunlight from the east, west, and south.

```xml
<type>
  WeatherStationCapability @capabilityType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:capability:WeatherStationCapability_{{SUFFIX}}") {
    getOutdoorTemperature: GetOutdoorTemperature @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetOutdoorTemperature_{{SUFFIX}}")
    getIlluminanceOfSun_south: GetIlluminanceOfSun_south @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_south_{{SUFFIX}}")
    getIlluminanceOfSun_west: GetIlluminanceOfSun_west @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_west_{{SUFFIX}}")
    getIlluminanceOfSun_east: GetIlluminanceOfSun_east @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_east_{{SUFFIX}}")
    getLight: GetLight @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetLight_{{SUFFIX}}")
    getWindSpeed: GetWindSpeed @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetWindSpeed_{{SUFFIX}}")
    getRainCheck: GetRainCheck @action(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetRainCheck_{{SUFFIX}}")
  }
</type>
```

Implementing your device's capability

Now you need to define the actions that you included in your device model's capability. To do this, look at your product specification to determine how your device performs the actions in the capability.

The device functions described in the Elsner P03 Weather Station specification map to the actions that you included in the device model capability.

- The three action types for getIlluminanceOfSun correspond with the device's three brightness sensors.
- The getLight action corresponds with its recognition of twilight and dawn.
- The getRainCheck action corresponds with the precipitation recognition function, and the getOutdoorTemperature and getWindSpeed actions correspond with the device's temperature and wind strength measurement functions.

The specification also contains a table named **P03-Modbus output string to the master**. This table contains the following information:

- The register address where each measurement is stored.
- The number of bytes required to store each measurement.
- The type of measurement stored at each register (under the **Variable** column).
- The **Meaning** of each measurement. This column also gives important information about the data type of the measurement value.
The following extract from the table can help you understand how the information in the specification maps both to your action definitions and to the device definition that you create when the abstract device model is complete.

<table>
<thead>
<tr>
<th>Byte number</th>
<th>Register address</th>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>Outdoor Temperature</td>
<td>H  with sign, value/10 = temperature xx.x °C</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Outdoor Temperature</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Sun sensor, south</td>
<td>H  1...99 Kilolux</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Sun sensor, south</td>
<td>L</td>
</tr>
</tbody>
</table>

This extract tells you that the temperature measurement is stored at register address 0, and its size in bytes is 2. It contains a sign (for above and below 0) and uses the Celsius scale. You know from this that you can use the built-in TDM data type Int16 as the return type of the getOutdoorTemperature action. You also know that you need to read 2 bytes to get the value. The extract also tells you that the device stores the south sunlight measurement at register address 1, and that its size is also 2 bytes. This value can also be returned as Int16.

In fact, all but one of the values that the actions measure can be returned with the Int16 data type. The rain sensor measurement (at register address 7) stores a 1-byte Boolean value.

When you create the action definitions, you use your understanding of the data types to be returned. When you create the device definition, you use the information about register addresses, data types, and sizes in bytes.

You can now define the following actions with your knowledge of the device, what it does, and how it performs its actions.

```plaintext
type GetOutdoorTemperature @actionType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetOutdoorTemperature_{{SUFFIX}}") {
  return: DegreesCelsius @property(id: "urn:tdm:aws:Property:Int16")
}
type GetIlluminanceOfSun_south @actionType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_south_{{SUFFIX}}") {
  return: KiloLux @property(id: "urn:tdm:aws:Property:Int16")
}
type GetIlluminanceOfSun_west @actionType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_west_{{SUFFIX}}") {
  return: KiloLux @property(id: "urn:tdm:aws:Property:Int16")
}
type GetIlluminanceOfSun_east @actionType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetIlluminanceOfSun_east_{{SUFFIX}}") {
  return: KiloLux @property(id: "urn:tdm:aws:Property:Int16")
}
type GetLight @actionType(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:action:GetLight_{{SUFFIX}}") {
```
Creating the definition for your device

You now have a complete abstract device model. You've defined what capability the weather station performs and the actions that it takes to fulfill this capability. Now you need to define how your device implements the abstract device model's capability.

To do this, you must know the following:

- The Modbus endpoint ID to which AWS IoT Things Graph sends messages
- The address of each register from which the master device gets each piece of information returned by each action
- The number of bytes to read
- The data type of each measurement value

The following example shows how these values are used in a device definition. (For more information about the elements of a Modbus device definition, see Modbus.)

```query
ModbusElsnerWeatherStation @device(id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:device:ModbusElsnerWeatherStation_{{SUFFIX}}", deviceModel: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:deviceModel:WeatherStation_{{SUFFIX}}") {
  Modbus(ServerId:"2") {
    Capability (id: "urn:tdm:{{REGION}}/{{ACCOUNT_ID}}/default:capability:WeatherStationCapability_{{SUFFIX}}") {
      Action(name: "getOutdoorTemperature") {
        Params
        ReadInputRegisters {
          Request (Address: 0, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
          }
        }
      }
      Action(name: "getIlluminanceOfSun_south") {
        Params
        ReadInputRegisters {
          Request (Address: 1, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
          }
        }
      }
      Action(name: "getIlluminanceOfSun_west") {
        Params
```
Service modeling 101

This topic describes what you need to think about and plan for when you're creating models for the services that you want to include in your flows. We use two example services, one that exposes a capability of the Amazon Rekognition service and another that implements an AWS Lambda function.
The approaches that this topic takes to modeling services can be applied to other AWS services and Lambda functions.

The ServiceModels101.zip file contains all of the GraphQL code discussed in this topic.

Note
The Amazon Rekognition and getS3Lambda service models discussed in this topic are available in the AWS IoT Things Graph console. The example in Creating a flow with Lambda functions uses the getS3Lambda service discussed here. The example in Creating a flow with devices and a service uses the Amazon Rekognition service discussed in this topic.

Modeling an AWS service

Amazon Rekognition is a machine learning service that can identify objects, people, text, scenes, and activities. It supports REST requests to endpoints that expose each of its capabilities. The example in this topic implements the Amazon Rekognition DetectFaces API as an AWS IoT Things Graph service.

Before you get started writing your model, you need to understand the requirements of the underlying AWS service. You can find this information in the Amazon Rekognition API reference.

Understand the underlying service

The following table from the first page of the Amazon Rekognition API Reference contains the headers that you need to add to every HTTP operation in the API.

<table>
<thead>
<tr>
<th>Header</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Type:</td>
<td>application/x-amz-json-1.1</td>
<td>Specifies that the request content is in JavaScript Object Notation (JSON). Also specifies the JSON version.</td>
</tr>
<tr>
<td>X-Amz-Date:</td>
<td>&lt;Date&gt;</td>
<td>The date used to create the signature in the Authorization header. The format must be ISO 8601 basic in the YYYYMMDD'T'HHMMSS'Z' format. For example, the following date/time 20141123T120000Z is a valid x-amz-date for use with Amazon Rekognition.</td>
</tr>
<tr>
<td>X-Amz-Target:</td>
<td>RekognitionService.&lt;operation&gt;</td>
<td>The target Amazon Rekognition operation. For example, use RekognitionService.ListCollections to call the ListCollections operation.</td>
</tr>
</tbody>
</table>

Your service model must include these parameters in the calls that it makes to the DetectFaces API. The AWS IoT Things Graph runtime adds the X-Amz-Date header to your requests automatically, so your model doesn’t have to account for that. Therefore, your HTTP request definition will include the following header parameters.

```javascript
headerParams {
    param(name:"Accept", property:"urn:tdm:aws:Property:String", value:"application/json")
}```
The headerParams block sets the required header parameters of the request. Amazon Rekognition requires you to specify the x-amz-json-1.1 content type and the REST endpoint (X-Amz-Target) for the capability that you want to use (RekognitionService.DetectFaces). In this example, the AWS IoT Things Graph service expects a return value of json, so you also include an Accept header.

Next, you need to look at the reference documentation for the DetectFaces API. This API requires you to pass an image as either a BLOB (a Base64-encoded binary data object) or as an S3Object. This example service passes an Amazon S3 object, so the HTTP request definition includes the following body parameters.

The bodyParams block can define the name-value pairs separately, or you can pass the name-value pairs in a JSON object. This example uses a JSON object, which is why the name attribute is an empty string. The $macro function replaces placeholders wrapped inside curly braces and prepended by dollar signs (${}) with variables whose values are supplied by the service activity. The service that we're defining requires the Amazon S3 bucket name and item name (also called the S3 object key name) to be supplied as parameters to its activity.

Define the action

Now that you understand the requirements of the AWS service that you're implementing as an AWS IoT Things Graph service, you can define the action or actions to implement in your service model.

The following GraphQL shows how to define the DetectFaces action.

The @actionType directive assigns a URN value that uniquely identifies the new action. The DetectFaces action consists of two string properties that set the Amazon S3 bucket and item names. The return keyword assigns a name to the return value (RkgnResponseProperty) and sets its property type to JSON.

Define the capability

Like a device, an AWS IoT Things Graph service implements a capability. Your service capability consists of the actions that you defined for it. The following GraphQL defines an Amazon Rekognition capability that contains the DetectFaces action that you just defined.
Define the service model

At this point you have the pieces you need to create the full service model. Your service model implements the RekognitionCap capability that you just created. You enable interaction with the service by specifying REST as the communication protocol inside the service definition. We've already seen the pieces of the service model that set the header and body parameters of the HTTP request to the REST service. You just need to put all of the pieces together and make sure that the appropriate parameters are passed to the device action.

The REST keyword inside the @service declaration defines this as an HTTP AWS IoT Things Graph service that uses a REST interface to make calls to the underlying AWS service. The Action name corresponds with the name of the action in the RekognitionCap capability definition.

The HttpPost block contains the Request and Response definitions. We've already seen the headerParams and bodyParams blocks. The #macro function used to compose the HTTP request URL substitutes the ${systemConfig.awsRegion} placeholder with the user's AWS Region. (You can also use $systemConfig.accountId when you need to retrieve the user's account ID.)

If the underlying web service is an AWS service, the Request block also requires values for auth and awsServiceName. Signature Version 4 (SigV4) is how AWS authenticates information sent in HTTP requests. Currently authentication is supported only for AWS services.
The `Response` block sets the content type of the HTTP response as a JSON object. This corresponds with the return type that you specified in the `DetectFaces` action definition.

The `params` block contains the two parameters that must be sent to the service when it’s used in a flow. In Creating a flow with devices and a service, the `WebServiceActivity` definition for the Amazon Rekognition service model (which you add when you’re creating the flow) retrieves these values from the CameraRkgnExample device.

```json
{  
  WebserviceActivity(webservice: "urn:tdm:aws/examples:service:Rekognition", out: "rekognitionResult") {  
    detectFaces(bucketName: "#{cameraResult.s3BucketName}", itemName: "#{cameraResult.s3ItemName}"
  }
}
```

The `WebServiceActivity` definition expects the camera to send the image as an Amazon S3 bucket. Your own flow might need to work differently, and your camera might not be able to pass images in this way. If your camera sends images in another format, your `DetectFaces` action will need to contain other properties to account for the differences.

The example in Creating a flow with devices and a service doesn't do anything with the Amazon Rekognition response. It simply passes the image to the screen device. You could fill in this gap by writing an AWS Lambda function that processes the results and applies logic before sending the image to the screen. You could include this function in your flow if you implement it as an AWS IoT Things Graph service. The following section shows you how to do that.

## Modeling an AWS Lambda function

An AWS IoT Things Graph service that implements a Lambda function has most of the same components as one that implements an AWS service. Because you write the underlying Lambda function, you have more control over what values it can take as inputs and return as outputs. The `getS3Lambda` service discussed in this example is used by the example in Creating a flow with Lambda functions. The code for this function is available for download in the Lambdas.zip file. You can use the sample code in that file to get started working with Lambda functions as AWS IoT Things Graph services.

### Model the Lambda function as an action

The `getS3Lambda` service implements a Lambda function with the same name. Like the Amazon Rekognition service discussed in the preceding section, this function also takes Amazon S3 bucket and object names as parameters. In this case, the parameters are named `bucket` and `key`, as in the following snippet from the function.

```javascript
const params = {
    Bucket: bucket,
    Key: key,
};
```

The service model definition needs a corresponding `params` section in its action block.

```json
params {
  param(name: "bucket", property:"urn:tdm:aws:property:String")
}
```
The `getS3ObjectAsString` code fetches the object from the bucket and returns an object that contains its value as a string and the length of the string. The following GraphQL shows the corresponding action definition.

```graphql
type getS3ObjectAsStringAction @actionType(id: "urn:tdm:aws/examples:action:getS3ObjectAsStringAction") {
  bucket: String @property(id: "urn:tdm:aws:property:String"),
  key : String @property(id: "urn:tdm:aws:property:String"),
  return : getS3ObjectAsStringResponse @property(id: "urn:tdm:aws/examples:property:getS3ObjectAsStringResponse")
}
```

The `@actionType` directive assigns a URN value that uniquely identifies the new action. The `getS3ObjectAsStringAction` action consists of two string properties that set the Amazon S3 bucket and object names. The `return` keyword assigns a name to the return value (`getS3ObjectAsStringResponse`) and sets its property type to `urn:tdm:aws/examples:property:getS3ObjectAsStringResponse`. This is not a built-in property type. It’s a complex property that contains both the string and the length. You need to define this property type.

**Define a complex property**

Complex properties are created as instances of states. States, in turn, consist of the properties that you want to include in your complex property.

You begin by creating a state that contains string and `Int32` values that represent the content of the Amazon S3 object and the length of the string.

```graphql
type S3ObjectDocument @stateType(id : "urn:tdm:aws/examples:State:S3ObjectDocument") {
  message: String @property(id: "urn:tdm:aws:property:String"),
  length: Int @property(id: "urn:tdm:aws:Property:Int32")
}
```

This state consists of a String property named `message` and an `Int32` property named `length`. Now you can create a complex property that is an instance of this state.

```graphql
type getS3ObjectAsStringResponse @propertyType(id: "urn:tdm:aws/examples:property:getS3ObjectAsStringResponse" instanceOf: "urn:tdm:aws/examples:State:S3ObjectDocument") {ignore: void}
```

**Define the capability**

Like a device, an AWS IoT Things Graph service implements a capability. Your service capability consists of the actions that you defined for it. The following GraphQL defines a capability that contains the `getS3ObjectAsStringAction` action that you just created.
Define the service model

At this point you have the pieces you need to create the full service model. Your service model implements the `getS3Capability` capability that you just created. We've already seen the parts of the service model that send the bucket and key values to the service. You just need to put everything together and ensure that the appropriate parameters are passed to the service action.

You enable interaction with the Lambda function by specifying `AwsLambda` as the communication protocol inside the service definition. If your Lambda function runs in an AWS IoT Greengrass group, specify `InvokeGreengrassLambda` inside the `Action` block. If your Lambda function runs in the cloud, specify `InvokeCloudLambda` instead.

```plaintext
query getS3Lambda @service(id: "urn:tdm:aws/examples:Service:getS3Lambda") {
  AwsLambda {
    getS3Capability(id: "urn:tdm:aws/examples:capability:getS3Capability") {
      Action(name: "getS3ObjectAsString") {
        params {
          param(name: "bucket", property:"urn:tdm:aws:property:String")
          param(name: "key", property:"urn:tdm:aws:property:String")
        }
        InvokeGreengrassLambda {
          Request(arn:"$macro(arn:aws:lambda:${systemConfig.awsRegion}:
          ${systemConfig.awsAccountId}:function:GetS3Object:1)") {
            params {
              param(name: "bucket", property:"urn:tdm:aws:property:String",
                value: "${bucket.value}"
              param(name: "key", property:"urn:tdm:aws:property:String",
                value: "${key.value}")
            }
            Response {
              responsePayload(property: "urn:tdm:aws/
                examples:property:getS3ObjectAsStringResponse")
            }
          }
        }
      }
    }
  }
}
```

The `AwsLambda` keyword inside the `@service` declaration defines this as an AWS IoT Things Graph service that implements an AWS Lambda function. The `Action` name corresponds with the name of the action in the `getS3Capability` capability definition.

The `InvokeGreengrassLambda` block contains the `Request` and `Response` definitions. The `$macro` function used to compose the ARN of the Lambda function substitutes the `${systemConfig.awsRegion}` and `${systemConfig.accountId}` placeholders with the user's AWS Region and account ID. This example service assumes that the user has created a Lambda function named `GetS3Object`. It also requires the user to use version 1 of that function.

You can use a Lambda function that runs in the cloud with the `InvokeCloudLambda` block, which works in the same way.
Creating and uploading entities

Creating entities for a flow involves three main tasks:

- Defining the entities (including the devices) in the flow.
- Uploading the entities to your private namespace.
- Associating things in your registry with the devices that you've defined and uploaded.

This topic describes how to complete these tasks by using either the AWS CLI or the AWS IoT Things Graph console.

**Step 1. Define the entities**

The following GraphQL contains a complete definition of entities that compose the flow described in Working with flows. (For an overview of how GraphQL is used in the Things Graph Data Model (TDM), see AWS IoT Things Graph data model and GraphQL.

These definitions create a barcode reader and another device that does something with the reader output, with all of the TDM entities that compose the devices.

**Note**

Before uploading your models, you must wrap all of your definitions between opening and closing curly braces, as in the following example.

```graphql
#
# Basic type definitions

type String @propertyType(id: "urn:tdm:REGION/ACCOUNT ID/default:property:String", dataType: String) {
  IGNORE: VOID
}

type
```
Boolean @propertyType(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Boolean", dataType: Boolean) {
    IGNORE: VOID
}

# Complex Type (State) definitions
type BarcodeType @stateType(id: "urn:tdm:REGION/ACCOUNT ID/default:state:BarcodeType") {
    id: Property @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:String")
}

type Barcode @propertyType(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode", instanceOf: "urn:tdm:REGION/ACCOUNT ID/default:state:BarcodeType") {
    IGNORE: VOID
}

# Action definitions
type DeviceA_ReadBarcode @actionType(id: "urn:tdm:REGION/ACCOUNT ID/default:action:DeviceA_ReadBarcode") {
    return: Property @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode")
}

type DeviceB_Action @actionType(id: "urn:tdm:REGION/ACCOUNT ID/default:action:DeviceB_Action") {
    input: Property @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode")
}

# Capability definitions
type DeviceA @capabilityType(id: "urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceA") {
    readBarcode: Action @action(id: "urn:tdm:REGION/ACCOUNT ID/default:action:DeviceA_ReadBarcode")
}

type DeviceB @capabilityType(id: "urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceB") {
    doSomething: Action @action(id: "urn:tdm:REGION/ACCOUNT ID/default:action:DeviceB_Action")
}

# Device model definitions (abstract device)
type DeviceB @deviceModel(id: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB", capability: "urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceB") {
    IGNORE: VOID
}

type DeviceA @deviceModel(id: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA", capability: "urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceA") {
    IGNORE: VOID
}

# Device definitions (physical device)
query DeviceA @device(id: "urn:tdm:REGION/ACCOUNT ID/default:device:DeviceA", deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA") {
    MQTT {
        Capability(id: "urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceA") {
            Action(name: "readBarcode") { 
                Publish {
                    Request(topic: "DeviceA/in") {
                        params
                    }
                }
            }
        }
    }
}
These models create the properties, states, actions, and capabilities, and the two devices that contain these other entities.

**Step 2. Upload the entity definitions**

After you have finished writing the entity definitions that compose your flow, you upload the definitions to your private namespace. You can do this in two ways, by using the UploadEntityDefinitions API and the CLI or by using the AWS IoT Things Graph console.

**Upload the entity definition (CLI)**

1. **Create the JSON definition document.**

   The UploadEntityDefinitions API consumes a JSON object with two parameter values: language and text.

   Currently, the only supported value for language is GraphQL. The value of text is a set of TDM definitions implemented in GraphQL. The resulting JSON looks like the following example.

   ```json
   {
     "language": "GRAPHQL",
     "text": "string containing the GraphQL models"
   }
   ```
You can construct the JSON payload in any way that your development framework and language support. For convenience, the following example contains a valid JSON object that you can copy into a file if you don't want to create one yourself.

```json
{
    "language": "GRAPHQL",
    "text": "

    type String @propertyType(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: String',
dataType: String) { IGNORE: VOID }
    type Boolean @propertyType(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: Boolean',
dataType: Boolean) { IGNORE: VOID }
    type BarcodeType @stateType(id: 'urn:tdm:REGION/ACCOUNT ID/default:state:BarcodeType') { id: Property @property(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: String') }
    type Barcode @propertyType(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: Barcode',
instanceOf: 'urn:tdm:REGION/ACCOUNT ID/default:state:BarcodeType') { IGNORE: VOID }
    type DeviceA_ReadBarcode @actionType(id: 'urn:tdm:REGION/ACCOUNT ID/default:action:DeviceA_ReadBarcode') { (return: Property @property(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: Barcode')) }
    type DeviceB_Action @actionType(id: 'urn:tdm:REGION/ACCOUNT ID/default:action:DeviceB_Action') { input: Property @property(id: 'urn:tdm:REGION/ACCOUNT ID/default:property: Barcode') }
    type DeviceA @capabilityType(id: 'urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceA') { readBarcode: Action @action(id: 'urn:tdm:REGION/ACCOUNT ID/default:action:DeviceA_ReadBarcode') }
    type DeviceB @capabilityType(id: 'urn:tdm:REGION/ACCOUNT ID/default:capability:DeviceB') { doSomething: Action @action(id: 'urn:tdm:REGION/ACCOUNT ID/default:action:DeviceB_Action') }


    }
}
```

2. **Upload the JSON definition document.**

After you construct the JSON payload that contains the GraphQL definitions, upload it to your namespace by using the following CLI command.

```
aws iotthingsgraph upload-entity-definitions --document file://entity file name
```

This command is executed asynchronously. Before using any of these entities, check whether the execution is complete by using the `get-upload-status` command, as in the following example.

```
aws iotthingsgraph get-upload-status --upload-id UPLOAD ID
```
Upload the entity definition (AWS IoT Things Graph console)

1. In the AWS IoT Things Graph console, select the menu icon at the upper left of the page. Choose Models.

   ![AWS IoT Things Graph](image)

   Things Graph

   Flows
   Deploy

   ▼ Prepare

   Things

   Models

2. On the Models page, choose Create model.

   ![Create model](image)

3. On the Create device model page, enter a name for your model. You can choose Upload a GraphQL file or Code in a GraphQL editor.

   If you choose to upload a GraphQL file, you can use the same JSON definition document that you created in Upload the Entity Definition (CLI) (p. 82).

   For this example, choose Code in a GraphQL editor. Then choose Next.
4. On the **Provide GraphQL** page, copy the GraphQL for your model into the editor. Then choose **Import**. Verify that your new model is included in the list on the **Models** page.
Step 3. Associate devices with things

Now that you've created two devices, you associate them with things in your registry. (If you haven't created things for your devices already, see Register a device in the registry for instructions on how to create things.)

Associate devices with things (CLI)

When you have two configured things to associate with your abstract devices, you can associate them with your devices by using the following CLI command.

```bash
aws iotthingsgraph associate-entity-to-thing --thing-name DeviceA --entity-id "urn:tdm:REGION/ACCOUNT_ID/default:device:DeviceA"
aws iotthingsgraph associate-entity-to-thing --thing-name DeviceB --entity-id "urn:tdm:REGION/ACCOUNT_ID/default:device:DeviceB"
```
Associate devices with things (AWS IoT Things Graph console)

1. On the AWS IoT Things Graph console home page, select the menu icon at the upper left of the page. Choose Models.

2. On the Things page, select the check box next to the AWS IoT thing from your registry to associate with a device that you created. Choose Associate.

3. On the Select device model page, choose the device to associate with your thing. Then choose Associate.

4. On the Things page, reload the page and verify that your device and thing are associated.
5. Repeat steps 2 and 3 for ThingB and any other things you want to associate with devices.

For more information about creating and managing a flow that contains these entities, see Creating and deploying flows.

**Versioning and entity modeling**

AWS IoT Things Graph enables you to create different versions of your namespace. If you change one or more of the entities in your namespace in ways that aren't compatible with existing deployments, AWS IoT Things Graph creates a new version of your namespace. Because each instance of a workflow (flow) is associated with a namespace version, existing deployments of a flow continue to work because they're associated with earlier versions of the namespace.

**Uploading entities**

The UploadEntityDefinitions API takes two optional parameters that enable you to create versions of your namespace. If you set the value of the deprecateExistingEntities parameter to true, AWS IoT Things Graph deprecates all of the existing entities in the current version of your namespace before it uploads the new entity definitions. This creates a new version of your namespace.

**Synchronizing with the public namespace**

If you set the value of the syncWithPublicNamespace parameter to true, AWS IoT Things Graph synchronizes your namespace with the latest version of the public namespace. Because this might introduce incompatibilities with the entity definitions that existing flow deployments use, AWS IoT Things Graph creates a new version of your namespace.

Existing deployments of flows that use deprecated or updated entity definitions continue to work because they're associated with earlier versions of the namespace.

If you add any entity definitions from your private namespace to the public namespace, you must update your own flows to use the version of the entity that's in the public namespace.

For more information about namespaces, see AWS IoT Things Graph namespaces.

**Viewing models and things in the AWS IoT Things Graph console**

The models section of the AWS IoT Things Graph console shows you all of the available services, devices, and device models in your namespace and in the public namespace. It also provides important information about each device and device model, including its definition and associated things. The things section of the console shows you all of the things in your AWS IoT registry. It also enables you to associate those things with devices in your namespace and in the public namespace.

This topic describes the features of the models and things sections and explains how to use them.
Models list

The main page of the models section contains a list of all of the devices and device models in your namespace and in the public namespace, as shown in the following screenshot.

The Models list page displays services, devices, and device models. It also shows their AWS IoT Things Graph Data Model URNs.

To create a model, choose Create Model. Follow the instructions to upload a file that contains all the components (properties, state, actions, events, and capability) of your service, device, or device model. You can also edit the model in a GraphQL editor in the browser, as shown in the following screenshot.

The MotionSensor device model definition appears in the browser-based GraphQL editor.

For more information about creating device definitions with the AWS IoT Things Graph Data Model, see the AWS IoT Things Graph Data Model Reference and Example device and service definitions.

Model details

To see details about each service, device, or device model, select the linked name in the Device class/subclass column. (You can also select the box next to the name, and then choose View details.) The default Details tab on the page that's displayed provides summary information about the definition, as shown in the following screenshot.

The Model details page contains the URN of the service, device, or device model. It also displays the properties, actions, and events in the definition.

If you're viewing a device model, the Device models and Associated things tabs also appear. The Device models tab displays all of the devices in your namespace and in the public namespace that implement the device model and its capability. The following screenshot shows one device that implements the MotionSensor device model.

The Device models section for the MotionSensor device model displays the HCSR501MotionSensor device.

If you're viewing a device model, the Associated things tab displays all of the things in your AWS IoT registry that are associated with the devices that implement the device model. You can also search for associated things from this page.

The Associated things tab for the MotionSensor device model displays the associated things and their ARNs.

Model definition

You can view the definition of a service, device, or device model in two ways. To download a file that contains the definition, choose Download model. To view the definition in the browser, choose View definition. The complete model definition appears in a new window.

View definition displays the MotionSensor definition, including its capability, state, event, and property.

When you select the Details tab, Edit model and Edit appear on the right side of the screen and are enabled. The two buttons perform the same operation. To edit the definition, choose Edit or Edit Model. Follow the instructions to upload a file containing updated versions of the components (properties, state, actions, events, and capability) of your service, device, or device model. You can also edit the model in a GraphQL editor in the browser.
The things section of the AWS IoT Things Graph console lists all of the things in your AWS IoT registry. You can use this page to associate each thing with a device in your namespace or in the public namespace.

After you associate a thing with a device, you can include that thing in a deployment that includes a flow that contains the device or a device model from which the device inherits. The following screenshot shows an example of the things list.

The MotionSensor1 thing is associated with the HCSR501MotionSensor device. The Associated entity ID column displays the AWS IoT Things Graph Data Model URN of each device.

To associate a thing with a device, select the thing and then choose Associate with model. The Select device model list appears. You can page through the list or use the Search device models box to find the device you want.

The HCSR501MotionSensor is selected in the Select device model window.

A thing can be associated with only one device. To dissociate a thing with a device, select the thing and then choose Dissociate with model.
Creating and deploying flows

After you successfully model and upload the entities that compose your workflows (flows), you can create, deploy, and work with them. For more information about creating and managing the entities in a flow, see Modeling entities.

Topics
- Creating and deploying flows (AWS IoT Greengrass) (p. 91)
- Creating and deploying flows (cloud) (p. 99)
- Revising flows (p. 107)
- Adding triggers to flows (p. 108)
- Working with choice nodes (p. 110)

Creating and deploying flows (AWS IoT Greengrass)

After you model a flow and upload all of the entities that it contains, you create and deploy the flow.

Prerequisites

Note
The AWS IoT Greengrass group and Amazon S3 bucket must be created in the same AWS Region. The AWS IoT Things Graph entities that you create must also be in the same Region as these resources.

- An AWS account
- An AWS IoT Greengrass core, version 1.7 or later
- An AWS IoT Greengrass IAM role that has access to your S3 bucket

Add this role to your Greengrass group. For information on how to configure IAM roles for Greengrass, see Configure IAM roles.

- Completion of the steps in Creating and uploading models

We provide procedures for creating and deploying flows using either the AWS CLI or the AWS IoT Things Graph console. Both the AWS CLI and the console use the two devices that you created in Creating and uploading models.

Note
You must create a thingsgraph directory at the root of your AWS IoT Greengrass core device. If this directory doesn't exist, deployments to your core device won't work.

Topics
Create and deploy a flow (CLI)

You've created the models that you'll use in your flow. The remaining AWS IoT Things Graph Data Model (TDM) entities to create are the Workflow, System, and Flow configuration.

1. Define the flow.

The following GraphQL contains a definition of a flow that sends a message from a barcode reader to another device.

```graphql
{ query BarcodeReaderFlow($device1Id: String, $device2Id: String) @workflowType(id: "urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow") { variables { barcode @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode") } steps { step(name: "DeviceA", outEvent: ["step1_done"]) { DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA", deviceId: "${device1Id}", out: "barcode") @device(id: "urn:tdm:REGION/ACCOUNT ID/default:device:DeviceA") { readBarcode } } step(name: "DeviceB", inEvent: ["step1_done"]) { DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB", deviceId: "${device2Id}") @device(id: "urn:tdm:REGION/ACCOUNT ID/default:device:DeviceB") { doSomething(input: "${barcode}") } } } }
```

2. Create the flow.

After you model the flow, you create it by using the `CreateWorkflowTemplate` API. This API consumes a JSON object that contains two parameter values: `language` and `text`.

Currently, the only supported value for `language` is GraphQL. The value of `text` is a set of TDM definitions implemented in GraphQL.

The resulting JSON looks like the following example.

```json
{ "language": "GRAPHQL", "text": "query BarcodeReaderFlow($device1Id: String, $device2Id: String) @workflowType(id: "urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow") { variables { barcode @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode") } steps { step(name: "DeviceA", outEvent: ["step1_done"]) { DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA", deviceId: "${device1Id}", out: "barcode") { readBarcode } } step(name: "DeviceB", inEvent: ["step1_done"]) { DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB", deviceId: "${device2Id}", out: "barcode") { doSomething(input: "${barcode}") } } } }
```
Create and deploy a flow (CLI)

ID/default:deviceModel:DeviceB', deviceId: '${device2Id}') {doSomething(input: '

After you construct the JSON payload that contains the GraphQL definition of the flow, you create the flow in your namespace by using the following CLI command.

```
aws iotthingsgraph create-flow-template --definition file://flow file name
```

3. Define the system.

In this step, you define a system that contains the flow you just created, and the devices that you modeled and associated with two things in your registry (as described in Getting started with models).

The following GraphQL contains a system that includes your flow and those two things. The URNs in the example are the unique identifiers of the device models that are associated with the things you created in Getting started with models.

```
{ type
  securitySystem @systemType(id: 'urn:tdm:REGION/ACCOUNT ID/
  default:system:BarcodeReaderFlow') {
    device1: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA')
    device2: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB')
    barcodeReaderFlow: Workflow @workflow(id: 'urn:tdm:REGION/ACCOUNT ID/
    default:Workflow:BarcodeReaderFlow')
  }
}
```

4. Create the system.

After you define the system, you create it by using the CreateSystemTemplate API. This API consumes a JSON object that contains two parameter values: language and text.

Currently, the only supported value for language is GraphQL. The value of text parameter is a set of TDM definitions implemented in GraphQL.

The resulting JSON looks like the following example.

```
{ "language": "GRAPHQL",
  "text": "{type securitySystem @systemType(id: 'urn:tdm:REGION/ACCOUNT ID/
  default:system:BarcodeReaderFlow') {
  device1: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA')
  device2: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB')
  barcodeReaderFlow: Workflow @workflow(id: 'urn:tdm:REGION/ACCOUNT ID/
  default:Workflow:BarcodeReaderFlow'))}
}
```

After you construct the JSON payload that contains the GraphQL definition of the flow, you create the flow in your namespace by using the following CLI command.
aws iotthingsgraph create-system-template --definition file://system file name

5. Define the flow configuration.

Systems are deployed within flow configurations.

The following GraphQL defines a flow configuration for the system that you just created. It passes a trigger to the flow that starts the flow every 10 seconds.

```graphql
{ query BarcodeReaderFlow @deployment(id: 'urn:tdm:REGION/ACCOUNT ID/default:deployment:BarcodeReaderFlow', systemId: 'urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow') {
  device1(deviceId: 'DeviceA')
  device2(deviceId: 'DeviceB')
  triggers {
    TenSecondTrigger(description: 'a trigger') {
      condition(expr: 'every 10 seconds')
      action(expr: 'ThingsGraph.startFlow("barcodeReaderFlow", bindings[name == "device1"].deviceId, bindings[name == "device2"].deviceId)')
    }
  }
}
}
```

6. Create the flow configuration.

After you define the flow configuration, you create it by using the `CreateSystemInstance` API. This API consumes a JSON object that contains two parameter values: `language` and `text`.

Currently the only supported value for `language` is GraphQL. The value of `text` is a set of TDM definitions implemented in GraphQL.

The resulting JSON looks like the following example.

```json
{
  "language": "GRAPHQL",
  "text": "{ query BarcodeReaderFlow @deployment(id: 'urn:tdm:REGION/ACCOUNT ID/default:deployment:BarcodeReaderFlow', systemId: 'urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow') { device1(deviceId: 'DeviceA') device2(deviceId: 'DeviceB') triggers { TenSecondTrigger(description: 'a trigger') { condition(expr: 'every 10 seconds') action(expr: 'ThingsGraph.startFlow("barcodeReaderFlow", bindings[name == "device1"].deviceId, bindings[name == "device2"].deviceId)') } } }
}
```

The `CreateSystemInstance` API also requires three additional parameters:

`target`

The target type for your deployment. For AWS IoT Greengrass deployments, specify `GREENGRASS`.

`s3BucketName`

The name of the Amazon S3 bucket where AWS IoT Things Graph will deploy the dependency closure of the flow configuration.
Create and deploy a flow (AWS IoT Things Graph console)

1. Sign in to the AWS IoT Things Graph console.

Choose Create flow.

2. Name the flow.

In the Flow configuration pane, enter a name for your flow. This name can't contain any spaces.
Choose **Create flow**.

3. **Add the trigger and devices to the flow.**

   On the **Logic** tab, choose **Clock**, and then drag it into the flow designer.

   Search for **DeviceA**, which is a barcode reader that you created in [Creating and uploading models](#). Select the device and drag it into the flow designer. Do the same for **DeviceB**.

   ![Flow Diagram](image)

4. **Connect the devices.**

   In the flow designer, select the edge of **ClockTrigger** and connect it to **DeviceA**. Connect **DeviceA** and **DeviceB** in the same way.

   ![Flow Diagram](image)

   The event appears between the two devices because **DeviceA** passes an output (the bar code) to **DeviceB**. You can pass this output to multiple devices and services by connecting this event to them, as in the following image.

   ![Flow Diagram](image)

   This is the correct way to pass output to multiple entities. Directly connecting one device or service to multiple devices or services causes a syntax error when you try to publish the flow.

5. **Update the ClockTrigger.**

   In the trigger editor that appears in the right pane, for **Frequency**, enter 10, and then select seconds from the menu on the right. For **Action**, choose **ThingsGraph.startFlow**.

6. **Update the device action for DeviceA.**

   In the flow designer, select **DeviceB**. Select **No action configured** in the action editor that appears in the right pane. Select **readBarcode** from the drop-down box that appears. Select the arrow that appears to the left of the **Output** option. Enter **deviceAResult** in the text box that appears under **Output**.

   Click the surface of the flow designer to close the action editor.

7. **Update the device action for DeviceB.**
In the flow designer, select **DeviceB**. Select **No action configured** in the action editor that appears in the right pane. Select **doSomething** from the drop-down box that appears. Select **Define Input** under the **Inputs** option. Enter \${deviceAResult} in the pop-up box that appears. Select the **Save Input** button

Click the surface of the flow designer to close the action editor.

8. **Publish the flow.**

Choose **Publish** at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed. Then choose **Go to flow list** to verify that your flow is created.

9. **Start creating the flow configuration.**

On the **Flows** list page, select the check box next to the flow that you just created, and then choose **Create flow configuration**.

10. **Name the flow configuration.**

On the **Describe flow configuration** page, select your flow and enter a deployment name. The deployment name can't contain spaces. Choose **Greengrass**, and then choose **Next**.

11. **Configure the target.**

On the **Configure target** page, enter the name of your Amazon S3 bucket and the AWS IoT Greengrass group to which your AWS IoT Greengrass core device belongs. Choose **Next**.
12. **Select things.**

    On the **Map Things** page, from the menu under **deviceA**, select the thing that you associated with this device in **Creating and uploading models**. Do the same for **deviceB**. Choose **Next**.

13. **View the trigger.**

    On the **Set up triggers** page, the following GraphQL appears in the editor. This GraphQL specifies the time intervals at which the flow runs. This flow runs every 10 seconds. You don't need to edit this code.

    Choose **Review**.
14. **Review and create**

   On the **Review and create** page, review the information you entered for your flow configuration.

   Choose **Create**.

15. **Deploy**

   When the **Flow configuration created** message appears, choose **Deploy now**.

   After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

---

**Creating and deploying flows (cloud)**

After you model a flow and upload all of the entities that it contains, you create and deploy the flow.

**Prerequisites**
Create and deploy a flow (CLI)

You've created the models that you'll use in your flow. The remaining AWS IoT Things Graph Data Model (TDM) entities to create are the Workflow, System, and Flow configuration.

1. **Define the flow.**

   The following GraphQL contains a definition of a flow that sends a message from a barcode reader to another device.

   ```graphql
   query BarcodeReaderFlow($device1Id: String, $device2Id: String) @workflowType(id: "urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow") {
     variables {
       barcode @property(id: "urn:tdm:REGION/ACCOUNT ID/default:property:Barcode")
     }
     steps {
       step(name: "DeviceA", outEvent: ["step1_done"])
       DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA", deviceId: "#{device1Id}", out: "barcode") @device(id: "urn:tdm:REGION/ACCOUNT ID/default:device:DeviceA") {
         readBarcode
       }
       step(name: "DeviceB", inEvent: ["step1_done"])
       DeviceActivity(deviceModel: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB", deviceId: "#{device2Id}", out: "barcode") @device(id: "urn:tdm:REGION/ACCOUNT ID/default:device:DeviceB") {
         doSomething(input: "#{barcode}"
       }
     }
   }
   ```

2. **Create the flow.**

   After you model the flow, you create it by using the `CreateWorkflowTemplate` API. This API consumes a JSON object that contains two parameter values: `language` and `text`.

   Currently, the only supported value for `language` is GraphQL. The value of `text` is a set of TDM definitions implemented in GraphQL.
The resulting JSON looks like the following example.

```
{
  "language": "GRAPHQL",
  "text": "{query BarcodeReaderFlow($device1Id: String, $device2Id: String)
@workflowType(id: 'urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow')
  {variables {barcode @property(id: 'urn:tdm:REGION/ACCOUNT ID/default:property:Barcode')} steps {step(name: 'DeviceA', outEvent: ['step1_done'])
    {DeviceActivity(deviceModel: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA', deviceId: '${device1Id}', out: 'barcode') {readBarcode}} step(name: 'DeviceB', inEvent: ['step1_done'])
    {DeviceActivity(deviceModel: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB', deviceId: '${device2Id}') {doSomething(input: '${barcode}')}}}}}
```

After you construct the JSON payload that contains the GraphQL definition of the flow, you create the flow in your namespace by using the following CLI command.

```
aws iotthingsgraph create-flow-template --definition file://flow file name
```

3. **Define the system.**

In this step, you define a system that contains the flow you just created, and the devices that you modeled and associated with two things in your registry (as described in Getting started with models).

The following GraphQL contains a system that includes your flow and those two things. The URNs in the example are the unique identifiers of the device models that are associated with the things you created in Getting started with models.

```
{
  type
    securitySystem @systemType(id: "urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow") {
      device1: Thing @thing(id: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA")
      device2: Thing @thing(id: "urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB")
      barcodeReaderFlow: Workflow @workflow(id: "urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow")
    }
}
```

4. **Create the system.**

After you define the system, you create it by using the CreateSystemTemplate API. This API consumes a JSON object that contains two parameter values: language and text.

Currently, the only supported value for language is GraphQL. The value of text parameter is a set of TDM definitions implemented in GraphQL.

The resulting JSON looks like the following example.

```
{
  "language": "GRAPHQL",
```
"text": "\n\n{type securitySystem @systemType(id: 'urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow')
  {device1: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceA')
   device2: Thing @thing(id: 'urn:tdm:REGION/ACCOUNT ID/default:deviceModel:DeviceB')
   barcodeReaderFlow: Workflow @workflow(id: 'urn:tdm:REGION/ACCOUNT ID/default:Workflow:BarcodeReaderFlow')}\n}\n\n" text": \n\nAfter you construct the JSON payload that contains the GraphQL definition of the flow, you create the flow in your namespace by using the following CLI command.

```
aws iotthingsgraph create-system-template --definition file://system file name
```

5. **Define the flow configuration.**

Systems are deployed within flow configurations.

The following GraphQL defines a flow configuration for the system that you just created. It passes a trigger to the flow that starts the flow every 10 seconds.

```graphql
\n{ query BarcodeReaderFlow @deployment(id: 'urn:tdm:REGION/ACCOUNT ID/default:deployment:BarcodeReaderFlow', systemId: 'urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow')
  {device1(deviceId: 'DeviceA')
   device2(deviceId: 'DeviceB')
   triggers {
     TenSecondTrigger(description: 'a trigger') {
       condition(expr: 'every 10 seconds')
       action(expr: 'ThingsGraph.startFlow("barcodeReaderFlow", bindings[name == "device1"].deviceId, bindings[name == "device2"].deviceId)')
     }
   }
}\n\n```

6. **Create the flow configuration.**

After you define the flow configuration, you create it by using the CreateSystemInstance API. This API consumes a JSON object that contains two parameter values: language and text.

Currently the only supported value for language is GraphQL. The value of text is a set of TDM definitions implemented in GraphQL.

The resulting JSON looks like the following example.

```json
{
  "language": "GRAPHQL",
  "text": "\n{ query BarcodeReaderFlow @deployment(id: 'urn:tdm:REGION/ACCOUNT ID/default:deployment:BarcodeReaderFlow', systemId: 'urn:tdm:REGION/ACCOUNT ID/default:system:BarcodeReaderFlow')
  {device1(deviceId: 'DeviceA')
   device2(deviceId: 'DeviceB')
   triggers { TenSecondTrigger(description: 'a trigger') {
     condition(expr: 'every 10 seconds')
     action(expr: 'ThingsGraph.startFlow("barcodeReaderFlow", bindings[name == "device1"].deviceId, bindings[name == "device2"].deviceId)')
   }
  }
}\n\n"}
```
For cloud deployments, the `CreateSystemInstance` API also requires two additional parameters:

- `target`
  
  The target type for your deployment. For cloud deployments, specify `CLOUD`.

- `flowActionsRoleArn`

  The ARN of the IAM role that AWS IoT Things Graph assumes when it executes the flow. This role must have read and write access to Lambda and AWS IoT and any other AWS services that the flow uses when it executes.

After you construct the JSON payload that contains the GraphQL definition of the flow configuration, upload it to your namespace by using the following CLI command. The output of this command includes an ID for your flow configuration. You use this value when you deploy the flow configuration.

```bash
aws iotthingsgraph create-system-instance --definition file://flow configuration file name --target CLOUD --flow-actions-role-arn "IAM Role"
```

7. **Deploy the flow configuration.**

   Deploy your flow configuration to the cloud by using the `DeploySystemInstance` API.

   The following command shows how to deploy a flow configuration by using the AWS CLI.

   ```bash
   aws iotthingsgraph deploy-system-instance --id SYSTEM DEPLOYMENT CONFIGURATION ID
   ```

---

**Create and deploy a flow (AWS IoT Things Graph console)**

1. **Sign in to the AWS IoT Things Graph console.**

   Choose **Create flow.**

   ![Create new flow](image)

2. **Name the flow.**

   In the **Flow configuration** pane, enter a name for your flow. This name can't contain any spaces.

   Choose **Create flow.**
3. **Add the trigger and devices to the flow.**

On the Logic tab, choose Clock, and then drag it into the flow designer.

Search for **DeviceA**, which is a barcode reader that you created in *Creating and uploading models*. Select the device and drag it into the flow designer. Do the same for **DeviceB**.

4. **Connect the devices.**

In the flow designer, select the edge of **ClockTrigger** and connect it to **DeviceA**. Connect **DeviceA** and **DeviceB** in the same way.

The event appears between the two devices because **DeviceA** passes an output (the bar code) to **DeviceB**. You can pass this output to multiple devices and services by connecting this event to them, as in the following image.

This is the correct way to pass output to multiple entities. Directly connecting one device or service to multiple devices or services causes a syntax error when you try to publish the flow.

5. **Update the ClockTrigger.**

In the trigger editor that appears in the right pane, for **Frequency**, enter 10, and then select seconds from the menu on the right. For **Action**, choose **ThingsGraph.startFlow**.

6. **Update the device action for DeviceA.**

In the flow designer, select **DeviceB**. Select **No action configured** in the action editor that appears in the right pane. Select **readBarcode** from the drop-down box that appears. Select the arrow that appears to the left of the **Output** option. Enter **deviceAResult** in the text box that appears under **Output**.

Click the surface of the flow designer to close the action editor.

7. **Update the device action for DeviceB.**

In the flow designer, select **DeviceB**. Select **No action configured** in the action editor that appears in the right pane. Select **doSomething** from the drop-down box that appears. Select **Define Input**
under the Inputs option. Enter \${deviceAResult} in the pop-up box that appears. Select the Save Input button.

Click the surface of the flow designer to close the action editor.

8. **Publish the flow.**

Choose Publish at the upper right of the page. This creates the flow and adds it to the list of flows that can be deployed. Then choose Go to flow list to verify that your flow is created.

9. **Start creating the flow configuration.**

On the Flows list page, select the check box next to the flow that you just created, and then choose Create flow configuration.

10. **Name the flow configuration.**

On the Describe flow configuration page, select your flow and enter a deployment name. The deployment name can't contain spaces. Choose Cloud, and then choose Next.

11. **Configure the target.**

On the Configure target page, enter the ARN of the role that you created in Prepare for cloud deployments. Choose Next.

12. **Select things.**
On the **Map Things** page, from the menu under **deviceA**, select the thing that you associated with this device in *Creating and uploading models*. Do the same for **deviceB**. Choose **Next**.

13. **View the trigger.**

On the **Set up triggers** page, the following GraphQL appears in the editor. This GraphQL specifies the time intervals at which the flow runs. This flow runs every 10 seconds. You don’t need to edit this code.

Choose **Review**.
14. **Review and create**

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.

15. **Deploy**

When the **Flow configuration created** message appears, choose **Deploy now**.

After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.

### Revising flows

After you deploy a flow, you might want to revise it. Each flow is associated with a namespace version.

When you revise or create a flow, you should specify a compatible namespace version. This ensures that the flow continues to work even if the entities in your namespace are revised or deprecated.
When you pass the optional `compatibleNamespaceVersion` parameter to the `CreateFlowTemplate` API, you associate the flow with a specific version of the namespace. This ensures that deployments of the flow continue to work even if the entities in the namespace are deprecated or updated in ways that are incompatible with the flow.

The `UpdateFlowTemplate` also takes the `compatibleNamespaceVersion` parameter. You can use `compatibleNamespaceVersion` to associate the updated version of the flow with a version of the namespace that is compatible with your flow updates.

Because flows aren't versioned, you can't update earlier implementations of a flow. You can revise only the latest version of a flow.

For more information about namespaces, see AWS IoT Things Graph namespaces.

---

**Adding triggers to flows**

Triggers enable you to include expressions in your flows. You can create triggers that are time based, or that are tied to an event that occurs on a device.

You add time-based triggers to a flow using the Logic tab in the flow designer in the AWS IoT Things Graph console. A time-based trigger starts a flow after an interval of time that you define passes.

You can also use a device as a trigger if its capability contains at least one event. You specify a set of conditions that the event creates that triggers the flow.

This topic describes how to use the AWS IoT Things Graph console to include time-based and device triggers in your flows.

### Time-based triggers

To include a time-based trigger, on the Logic tab, choose Clock, and then drag it into the flow designer. This adds a node named `ClockTrigger` to the designer.

In the trigger editor that appears in the right pane, for Frequency, enter any number. Then from the menu on the right, select the type of time interval to use. Your choices are seconds, minutes, hours, days, weeks, months, and years.
Device triggers

When you include a device that has one or more events and no actions in its capability, the device is automatically a trigger, and the Set as Trigger switch in the right pane is disabled. This sort of device can only be a trigger and never a step in a flow.

When you use an event that has one or more actions and no events in its capability, the device is automatically a step and the Set as Trigger switch is disabled.

When you include a device that has one or more events and one or more actions, the device is a step by default, but the Set as Trigger switch is enabled. If you choose to set the device as a trigger, the trigger editor appears in the right pane. From the Condition event menu in this editor, select the event to use. For Action, choose ThingsGraph.startFlow. This is currently the only option for Action.

The following screenshot shows a button that has a ButtonPressed event in its capability. This event is selected under Condition event. The ButtonPressed event returns a payload that consists of one Boolean value named isLightOn. This value is stored in the lastEvent variable. The lastEvent variable stores values emitted by the trigger device's last event.
You can use the values stored in the `lastEvent` variable to write an expression that starts a flow whenever a condition is met. You enter this expression in the **Condition expression** field. In this case, the button's `ButtonPressed` event starts the flow when the value of `lastEvent.isLightOn` is `false`. To start the flow every time the event occurs, leave the **Condition expression** field blank.

You can write more complex expressions depending on the number and types of values emitted by the trigger event. For example, if a device emits a `lastEvent` variable that two properties named `type` and `value`, you could write the following expression.

```
lastEvent.type == "measurement" && lastEvent.value > 60
```

**Triggers and flow configurations**

When you create a flow configuration, the trigger in the flow can't be updated. When you update a trigger in a flow, you must create and deploy a new flow configuration that contains the updated trigger. Triggers in existing flow configurations can't be updated.

**Working with choice nodes**

Choice nodes enable you to include conditional logic during the execution of a flow. This topic describes how to use the choice node editor in the AWS IoT Things Graph console.

This topic shows you how to use choice nodes to create a time-based flow that turns a light on and off depending on the time of day. It assumes that your namespace includes a service that implements an AWS Lambda function that returns the hour of the day as an `Int`. It also assumes that your namespace contains a device model for a light that has `TurnOn` and `TurnOff` actions.

This example is for demonstration purposes only. You can apply the approach in this example any workflow (flow) that contains the following elements.
Create the flow

The following instructions describe how to place all of the elements of the flow (devices, service, events, and choice node) into the flow designer.

1. Open the AWS IoT Things Graph console, and then choose Create flow.

2. Create a flow.

   In the Flow configuration pane, name your flow (such as LightFlow).
Create the flow

3. Add the elements to the flow.

On the **Logic** tab, choose **Clock**, and then drag it into the flow designer.

On the **Service** tab, search for your Lambda service. Choose the service and drag it into the flow designer.

On the **Logic** tab, choose **Choice**, and then drag it into the flow designer.

On the **Devices** tab, choose your device or device model, and then drag it into the flow designer. Drag this same device into the flow designer a second time.

The two instances of the device or device model represent different steps that can occur in the flow, depending on the choice node output.

4. Connect the flow elements.

In the flow designer, connect the **ClockTrigger** to the Lambda function. Connect the Lambda function to the choice node. Connect the choice node to both instances of the device.

The flow designer displays one event between the Lambda function and the choice node, and one event between the choice node and each instance of the device or device model. You can change the default name of the event that follows the Lambda function. The other events will be named after events that you create in the choice node.
Configure events and choice node logic

The following instructions describe how to configure the trigger logic, service output, and choice node logic in the flow.

1. Configure the clock trigger.

   In the flow designer, select the ClockTrigger. In the trigger editor that appears in the right pane, for Frequency, enter 1, and then select hours from the menu on the right. For Action, choose ThingsGraph.startFlow.

2. Configure the output of the Lambda service.

   In the flow designer, select the service. In the action editor that appears in the right pane, select No action configured. Select the action that generates the output for the choice node. Provide a variable name to store the output. The logic in the choice node uses the value or values in this output to create events.
3. Name the choice node.

In the flow designer, select the choice node. In the choice editor that appears in the right pane, enter a value for **Choice title**.

4. Configure the condition for the first rule in the choice node.

Expand **Rule A**. Optionally, enter a title for the rule.

Enter an **expression** that evaluates the output from the Lambda service. In this example, the rule evaluates the current hour to determine whether the current hour is at night: `$(getHourResult > 14)`.
5. Configure the event for the first rule in the choice node.

Choose Add event. Enter an event name. In this example, the rule creates an event named `isNight` whenever the expression in the condition evaluates to `true`. 
6. Configure the condition for the second rule in the choice node.

Choose Add rule. Expand Rule B. Optionally, enter a title for the rule.

Enter an expression that evaluates the output from the Lambda service. In this example, the rule evaluates the current hour to determine whether the current hour is in the daytime: 

\[ \text{getHourResult} \leq 14 \].
7. Configure the event for the second rule in the choice node.

Choose Add event. Enter an event name. In this example, the rule creates an event named isDay whenever the expression in the condition evaluates to true.
8. Configure the default event in the choice node.

Expand Default rule. This rule determines what happens when the output from the previous step is unexpected. You can create new events and variables for this rule. This example defaults to the isNight event.

Note
The default event is required.
9. Name the events connected to each instance of the device.

Choose one of the events that follow the choice node. Enter the name of one of events that you created in the choice node. Choose the other event, and enter the name of the other choice node event. When you start typing, the flow designer prompts you to autocomplete the events that you created in the choice node.
10. Configure the device actions.

Select the device or device model that is connected to the `isNight` event. Choose No action configured, and select the `turnOn` action.

Select the device or device model that is connected to the `isDay` event. Choose No action configured, and select the `turnOff` action.

In this example, the same device executes one of two possible steps, depending on the events that are emitted by the choice node. The same device can be used in multiple steps and perform different actions depending on the events emitted by the choice node. You can use choice nodes to execute more complex logic depending on the number of rules, the complexity of the expressions in the rules, and the number of variables that you create.

For more information about expressions in choice nodes, see Using conditional logic in choice nodes.

**Using conditional logic in choice nodes**

AWS IoT Things Graph Data Model expressions provide a way to express conditional logic in choice nodes. This topic shows you how to implement these expressions in the AWS IoT Things Graph console. It also shows you how to include variables and logical operators in expressions. This topic assumes that you’ve read Working with choice nodes.

**Topics**

- Adding choice nodes and rules to your flow (p. 120)
- Writing expressions in rules (p. 123)

**Adding choice nodes and rules to your flow**

A choice node follows one or more steps in a flow that generate output that the flow uses to make decisions about what steps happen next. The following image shows this sort of decision point.

**Example**

A clock triggers a service that checks the time of day every hour. A choice node decides whether to turn a light on or off depending on the current time.

**To add a choice node to your flow**

1. Choose the Logic tab on the right side of the flow designer.
2. Choose **Choice** and drag it into the flow designer.

3. Connect a step in the flow (represented by a device or service) to the choice node. The flow designer displays an event between the device or service and the choice node. You can replace the default name of this event.

4. Connect the choice node to two or more devices and services. These devices and services represent steps in the flow. The flow designer displays an event between the choice node and each of the steps to which you've connected the choice node. These events are named after events that you create in the choice node.

5. Select the choice node. The flow designer displays a choice activity editor in the right pane.
The choice activity editor displays a text box that you use to add an optional title. The editor also contains one rule named **Rule A** and a default rule. Choose **Add rule** to add more rules.

**Example**

The following image shows what the flow designer canvas looks like when you configure the choice node and connect it to the potential next steps (the light turns on or off).

Rules tie conditions to events. If a condition in the rule is met, the choice node fires an event specified in the rule. You name each event that follows the choice node after the events that you create in the rules. The default rule specifies which event to use when conditions in the other rules aren't met. The default rule is required. The events connect the choice node to conditional steps in the flow.
Writing expressions in rules

Each rule (except the default rule) must contain a condition, and you express each condition with an AWS IoT Things Graph Data Model expression. This section explains how to incorporate variables and logic in expressions.

Using variable values in rules

If your expression consists of a simple operation on a variable or path, you can access its value by using the name of the variable or path, as in the following examples.

```
${getHourResult < 14}
${getHourResultA > getHourResultB}
${image.width == 32}
```

The following table lists the data types of the values used in these expressions.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>getHourResult</td>
<td>Int</td>
</tr>
<tr>
<td>getHourResultA</td>
<td>Int</td>
</tr>
<tr>
<td>getHourResultB</td>
<td>Int</td>
</tr>
<tr>
<td>image.width</td>
<td>An Image object with a width property of type Int.</td>
</tr>
</tbody>
</table>

String concatenation

The following example shows how to concatenate strings inside expressions.

```
${message == "Current time: " + getHourResult}
```

The following table lists the data types of the values used in this expression.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>String</td>
</tr>
<tr>
<td>getHourResult</td>
<td>Int</td>
</tr>
</tbody>
</table>

Logical operators

Use the following operators for logical AND, OR, and NOT.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;&amp;</td>
</tr>
</tbody>
</table>
Using conditional logic in choice nodes

<table>
<thead>
<tr>
<th>Logic</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>!</td>
</tr>
</tbody>
</table>

### Example

\[
\begin{align*}
\${\text{image.height}} &= 32 \text{ && } \text{image.width} = 55 \\
\${\text{getHourResult}} &= 14 \text{ || } \text{!isDay} \\
\text{!isNight} &
\end{align*}
\]

The following table lists the data types of the values used in this expression.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>image.height</td>
<td>An Image object with a width property of type Int.</td>
</tr>
<tr>
<td>getHourResult</td>
<td>Int</td>
</tr>
<tr>
<td>isDay</td>
<td>Boolean</td>
</tr>
<tr>
<td>isNight</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Creating systems

A system is a collection of devices, services, and a workflow (flow) that interact with each other in an IoT system. A system consists of AWS IoT and AWS IoT Things Graph flows.

When you create a flow in the AWS IoT Things Graph console, the console creates a system for you. For example, the system in Creating a flow with devices and a service contains RekognitionFlow. The following AWS IoT Things Graph Data Model (TDM) code shows the underlying definition of the system created in that example. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID.

```typescript
import { TDM } from 'aws-iot-things-graph-client';

const systemType = TDM.types.SystemType;
const motionSensor = TDM.types.MotionSensor;
const cameraRkgnExample = TDM.types.CameraRkgnExample;
const screen = TDM.types.Screen;
const RekognitionFlow = TDM.types.RekognitionFlow;

const system = systemType({
  motionSensor: motionSensor({ id: 'urn:tdm:aws/examples:DeviceModel:MotionSensor' }),
  cameraRkgnExample: cameraRkgnExample({ id: 'urn:tdm:aws/examples:DeviceModel:CameraRkgnExample' }),
  screen: screen({ id: 'urn:tdm:aws/examples:DeviceModel:Screen' }),
})
```

In this system:

- **@systemType declaration** creates a system whose identifier is the TDM URN inside the parentheses.
- **motionSensor: MotionSensor, etc.** are used to assign the device or device models and the flow to names (on the left side of each colon) that the flow configuration uses. The flow configuration uses the flow name to start the flow. It uses the device and device model names to assign specific things in your registry to each device or device model. In the flow configuration, these names must match the names in the system exactly.
- **@thing declarations** specify the devices or device models in the system.
- **@workflow declarations** specify the flows in the system. You don't need to specify services in the system definition.
You can include more than one flow in a system (as long as you write the TDM yourself). This enables you to deploy more than one flow at a time.

The following command shows how to create a system by using the AWS CLI.

```bash
aws iotthingsgraph create-system-template --definition language=GRAPHQL,text="TDM System Definition"
```

For more information about creating a system programatically, see CreateSystemTemplate in the AWS IoT Things Graph API Reference.

The following section describes how to create flow configurations.

## Creating flow configurations (AWS IoT Greengrass)

A flow configuration specifies all of the elements and properties required to deploy a system in a physical location. The flow configuration consists of an AWS IoT Things Graph Data Model (TDM) deployment definition, with attributes that specify other required resources. These include the name of the Amazon Simple Storage Service (Amazon S3) bucket from which the system is deployed, and the name of the AWS IoT Greengrass group to which the system is deployed.

This section describes the structure of a flow configuration and shows how to create one with the AWS CLI and the AWS IoT Things Graph console.

### Creating a flow configuration in the AWS CLI

This section describes how to create the flow configuration used in Creating a flow with devices and a service by using the AWS CLI. See that topic for complete instructions on setting up your environment, creating things, creating the flow (and the system), and associating things with devices. For instructions on how to create a flow with the AWS CLI, see Creating and deploying flows.

### Defining the flow configuration

The following AWS IoT Things Graph Data Model (TDM) shows the underlying definition of the flow configuration created in Creating a flow with devices and a service. Replace the `REGION` and `ACCOUNT ID` placeholders with your AWS Region and account ID.

```graphql
query Lobby @deployment(id: "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby" systemId: "urn:tdm:REGION/ACCOUNT ID/default:System:RekognitionFlowSystem") {
  motionSensor(deviceId: "MotionSensor1")
  screen(deviceId: "Screen1")
  cameraRkgnExample(deviceId: "Camera1")

  triggers {
    MotionEventTrigger(description: 'a trigger') {
      condition(expr: "devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.value")
    }
  }
}
```
In this deployment definition:

- You create a deployment definition as a GraphQL query.
- The @deployment declaration tells AWS IoT Things Graph to create a deployment definition whose identifier is the TDM URN inside the parentheses.
- The next value inside the parentheses is the URN of the system that contains the workflows (flows) that deploy with this deployment.
- The content inside the braces begins by assigning the names of the devices or device models in the flows (as specified in the system definition) to things that are registered in your AWS IoT registry. Before you make this assignment in the TDM, use the AssociateEntityToThing API to associate each device with each thing you’re using in the deployment. The deviceId values are the names of the things in your registry.
- The triggers block contains one or more triggers. The MotionEventTrigger in this example consists of a condition that triggers the flow and the action that starts when the trigger condition is met.
- The condition uses two path expressions to identify a device that is used in a flow and one of that device’s events, with a predicate expression that determines whether the device has detected motion. If the lastEvent value sent by the motionSensor device is true, a non-empty string, or a numeric value other than zero (0), the devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.value expression evaluates to true. This signifies that a motion detected event has occurred. If you want the expression to evaluate to true every time the event is fired, use devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.
- The action uses the ThingsGraph.startFlow function, which initiates the specified flow. The flow name matches the name of the flow in the system definition. The bindings path expressions specify the names of the things that are used in the flow.

Now that you have a complete deployment definition, you can create the flow configuration by using the AWS CLI.

The values that follow the flow name inside the action can be any valid flow parameters. The following example triggers show some of the values and path expressions that you can use inside the action block of a trigger.
Creating a flow configuration in the AWS CLI

For more information about triggers, see [Trigger](#).

**Creating the flow configuration**

The following command shows how to create a flow configuration by using the AWS CLI.

```bash
aws iotthingsgraph create-system-instance --definition language=GRAPHQL,text="TDM Deployment Definition" \ 
--target GREENGRASS --greengrass-group-name AWS IoT Greengrass Group Name --s3-bucket-name Amazon S3 Bucket Name
```

- **target parameter** – Specifies the target type of the deployment. For AWS IoT Greengrass deployments, specify GREENGRASS.
- **greengrass-group-name parameter** – Specifies the name of the AWS IoT Greengrass group to which the flow configuration is deployed.
- **s3-bucket-name parameter** – Specifies the Amazon S3 bucket that's used to store and deploy the resource file of the flow configuration.

When the operation completes, the AWS CLI returns the following deployment summary (as a JSON object).

```json
{
   "summary": {
      "status": "PENDING_DEPLOYMENT",
      "greengrassGroupName": "AWS IoT Greengrass Group Name",
      "target": "GREENGRASS",
      "arn": "arn:aws:iotthingsgraph:REGION:ACCOUNT ID:default#Deployment#Lobby",
      "updatedAt": 1547245009.256,
      "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby",
      "createdAt": 1547245009.256
   }
}
```

In this deployment summary:

- The **status** value in the **summary** object is **PENDING_DEPLOYMENT** when a flow configuration is created.
- The **id** value in the **summary** block is the TDM URN of the flow configuration.

The following command uses the **id** value to deploy the flow configuration to the target.

```bash
aws iotthingsgraph deploy-system-instance --id Flow Configuration Id
```

For more information about creating a flow configuration programmatically, see [CreateSystemInstance](#) in the [AWS IoT Things Graph API Reference](#).
Creating a flow configuration in the AWS IoT Things Graph console

These instructions are a subset of the procedure for creating and deploying the flow in Creating a flow with devices and a service. See that topic for complete instructions on setting up your environment, creating things, creating the flow (and the system), and associating things with devices.

1. Create the flow configuration.

Select the menu icon at the upper left of the page, and then select Flows to return to the Flows page. On the Flows page, select the box next to the flow that you just created, and then choose Create flow configuration.

2. Name the flow configuration.

On the Describe flow configuration page, select your flow, and then enter a flow configuration name. The flow configuration name can’t contain spaces. Choose Greengrass, and then choose Next.
3. Configure the target.

On the **Configure target page**, enter the name of your Amazon S3 bucket and the AWS IoT Greengrass group to which your AWS IoT Greengrass core device belongs. Choose **Next**.
4. Select things for your deployment.

The **Map Things** page provides an interface for selecting the specific things that you'll include in your deployment. The menus under each device in your deployment contain all of the things that you associated with the device. Because you’re getting started, the menus for each device on this page will include only one thing (the thing that you’ve associated with each device).

On the **Map Things** page, under the *motionSensor* device, select the motion sensor thing that you created earlier. Select the camera and screen things for the *Camera* and *Screen* devices. Choose **Next**.

5. View the trigger.

On the **Define trigger** page, the GraphQL that defines the motion event trigger for the flow appears in the editor. When the motion sensor detects motion, the `ThingsGraph.startFlow` function initiates the flow. You don't need to edit this code.

Choose **Review**.
6. **Review and create.**

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.
Creating flow configurations (cloud)

A flow configuration specifies all of the components and properties required to deploy a system in a physical location. The flow configuration consists of an AWS IoT Things Graph Data Model (TDM)

7. Deploy.

When the Flow configuration created message appears, choose Deploy now.

After a successful deployment, the Deployments page displays Deployed in target in the Status column.
deployment definition. The attributes in the definition specify other required resources, such as the IAM role that AWS IoT Things Graph assumes when it executes the flow in the cloud.

This section describes the structure of a flow configuration and shows how to create one with the AWS IoT Things Graph console and the AWS CLI.

Topics
- Creating a flow configuration in the AWS CLI (p. 134)
- Creating a flow configuration in the AWS IoT Things Graph console (p. 136)

Creating a flow configuration in the AWS CLI

These instructions are similar to the procedure for creating and deploying the flow in Creating a flow with devices and a service. That topic contains complete instructions on setting up your environment, creating things, creating the flow (and the system), and associating things with devices. Because this is a cloud deployment, instead of creating and configuring a AWS IoT Greengrass group, follow the instructions in Prepare for cloud deployments to create the IAM role that AWS IoT Things Graph assumes when it executes the flow.

Defining the flow configuration

The following AWS IoT Things Graph Data Model (TDM) shows the underlying definition of the flow configuration created in Creating a flow with devices and a service. Replace the REGION and ACCOUNT ID placeholders with your AWS Region and account ID.

```graphql
query Lobby @deployment(id: "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby" systemId: "urn:tdm:REGION/ACCOUNT ID/default:System:RekognitionFlowSystem") {

  motionSensor(deviceId: "MotionSensor1")
  screen(deviceId: "Screen1")
  cameraRkgnExample(deviceId: "Camera1")

  triggers {
    MotionEventTrigger(description: 'a trigger') {
      condition(expr: "devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.value")
      action(expr: "ThingsGraph.startFlow('RekognitionFlow', bindings[name == 'cameraRkgnExample'].deviceId, bindings[name == 'screen'].deviceId)")
    }
  }
}
```

In this deployment definition:

- You create a deployment definition as a GraphQL query.
- The @deployment declaration tells AWS IoT Things Graph to create a deployment definition whose identifier is the TDM URN inside the parentheses.
- The next value inside the parentheses is the URN of the system that contains the workflow (also called a flow) or flows that deploy with this deployment.
- The content inside the braces begins by assigning the names of the devices or device models in the flows (as specified in the system definition) to things that are registered in your AWS IoT registry. Before you make this assignment in the TDM, use the AssociateEntityToThing API to associate each device with each thing you’re using in the deployment. The deviceId values are the names of the things in your registry.
• The triggers block contains one or more triggers. The MotionEventTrigger in this example consists of a condition that triggers the flow and the action that starts when the trigger condition is met.

• The condition uses two path expressions to identify a device that is used in a flow and one of that device's events, with a predicate expression that determines whether the device has detected motion. If the lastEvent value sent by the motionSensor device is true, a non-empty string, or a numeric value other than zero (0), the devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.value expression evaluates to true. This signifies that a motion detected event has occurred. If you want the expression to evaluate to true every time the event is fired, use devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent.

• The action uses the ThingsGraph.startFlow function, which initiates the specified flow. The flow name matches the name of the flow in the system definition. The bindings path expressions specify the names of the things that are used in the flow.

Now that you have a complete deployment definition, you can create the flow configuration by using the AWS CLI.

The values that follow the flow name inside the action can be any valid flow parameters. The following example triggers show some of the values and path expressions that you can use inside the action block of a trigger.

```
DataEmitterTrigger01(description: "trigger on integer events") {
  condition(expr: "devices[name == 'dataEmitter'].events[name == 'IntegerEvent'].lastEvent.val > 10")
  action(expr: "ThingsGraph.startFlow('exampleFlow1', devices[name == 'dataEmitter'].events[name == 'IntegerEvent'].lastEvent.val, False, 1.1, ((String)devices[name == 'dataEmitter'].events[name == 'IntegerEvent'].lastEvent.val).charAt(0))")
}

DataEmitterTrigger02(description: "trigger on json events") {
  condition(expr: "devices[name == 'dataEmitter'].events[name == 'JsonEvent'].lastEvent")
  action(expr: "ThingsGraph.startFlow('exampleFlow2', -100, True, -1.0000000001, devices[name == 'dataEmitter'].events[name == 'JsonEvent'].lastEvent")
}
```

For more information about triggers, see Trigger.

Creating the flow configuration

The following command shows how to create a flow configuration by using the AWS CLI.

```
aws iotthingsgraph create-system-instance --definition language=GRAPHQL,text="TDM Deployment Definition" \
--target CLOUD --flow-actions-role-arn IAM Role \
--metrics-configuration true/false
```

• target parameter – Specifies the target type of the deployment. For cloud deployments, specify CLOUD.
• flow-actions-role-arn parameter – Specifies the role that AWS IoT Things Graph assumes when it executes the flow in the cloud. For instructions on how to create this role, see Prepare for cloud deployments.
When the operation completes, the AWS CLI returns the following deployment summary (as a JSON object).

```
{
   "summary": {
      "status": "PENDING_DEPLOYMENT",
      "greengrassGroupName": "AWS IoT Greengrass Group Name",
      "target": "GREENGRASS",
      "arn": "arn:aws:iotthingsgraph:REGION:ACCOUNT ID:default#Deployment#Lobby",
      "updatedAt": 1547245009.256,
      "id": "urn:tdm:REGION/ACCOUNT ID/default:Deployment:Lobby",
      "createdAt": 1547245009.256
   }
}
```

In this deployment summary:

- The **status** value in the **summary** object is **PENDING_DEPLOYMENT** when a flow configuration is created.
- The **id** value in the **summary** block is the TDM URN of the flow configuration.

The following command uses the **id** value to deploy the flow configuration to the target.

```
aws iotthingsgraph deploy-system-instance --id Flow Configuration Id
```

For more information about creating a flow configuration programmatically, see CreateSystemInstance in the AWS IoT Things Graph API Reference.

**Creating a flow configuration in the AWS IoT Things Graph console**

These instructions are similar to the procedure for creating and deploying the flow in Creating a flow with devices and a service. That topic contains complete instructions on setting up your environment, creating things, creating the flow (and the system), and associating things with devices. Because this is a cloud deployment, instead of creating and configuring a AWS IoT Greengrass group, follow the instructions in Prepare for cloud deployments to create the IAM role that AWS IoT Things Graph assumes when it executes the flow.

1. Create the flow configuration.

   Select the menu icon at the upper left of the page, and then select **Flows** to return to the **Flows** page. On the **Flows** page, select the box next to the flow that you just created, and then choose **Create flow configuration**.
2. Name the flow configuration.

On the *Describe flow configuration* page, select your flow, and then enter a flow configuration name. The flow configuration name can’t contain spaces. Choose *Cloud*, and then choose *Next*.

3. Configure the target.

On the *Configure target page*, enter the ARN of the role that you created in *Prepare for cloud deployments*. Choose *Next*. 

---

137
4. Select things for your deployment.

The **Map Things** page provides an interface for selecting the specific things that you'll include in your deployment. The menus under each device in your deployment contain all of the things that you associated with the device. Because you're getting started, the menus for each device on this page will include only one thing (the thing that you've associated with each device).

On the **Map Things** page, under the **MotionSensor** device, select the motion sensor thing that you created earlier. Select the camera and screen things for the **Camera** and **Screen** devices. Choose **Next**.

5. View the trigger.

On the **Define trigger** page, the GraphQL that defines the motion event trigger for the flow appears in the editor. When the motion sensor detects motion, the `ThingsGraph.startFlow` function initiates the flow. You don't need to edit this code.
Choose **Review**.

On the **Review and create** page, review the information you entered for your flow configuration. Choose **Create**.
Creating a flow configuration in the AWS IoT Things Graph console

7. **Deploy.**

When the **Flow configuration created** message appears, choose **Deploy now**.

After a successful deployment, the **Deployments** page displays **Deployed in target** in the **Status** column.
Viewing flow configurations and flow executions in the AWS IoT Things Graph console

The flow configurations pages in the AWS IoT Things Graph console show you all of the deployments that you’ve created and their current statuses. This page also links to more detailed summary information about each deployment. This includes the date when the deployment was created, and the things assigned to each device or device model in the flow.

The deployment details page also provides a list of flow executions. Each flow execution links to a page that contains the messages that each flow execution sent. These messages provide information such as the time when each step begins executing, when it completes successfully, and when it fails.

This topic describes the features of the flow configurations and flow executions pages and how to access them.

Flow configurations

The main page of the flow configurations section lists all of the flow configurations that you've created. For each flow configuration, it also lists the ID (an AWS IoT Things Graph Data Model [TDM] URN), the status, and date created. For AWS IoT Greengrass deployments, it lists the name of the AWS IoT Greengrass group that is the deployment target of the flow configuration.

To see more detailed summary data about a flow configuration, click its name. The next page displays general configuration information about the flow configuration.

The Assigned Things section of this page also displays the things from your AWS IoT registry that you assigned to the devices or device models in the flow.
Flow executions

To see a list of flow executions for the flow configuration, choose the Flow executions tab. The Flow executions page contains a list that shows every execution of the flow after you deployed it. A flow execution begins every time a trigger starts a flow, and it ends at the successful or unsuccessful completion of the flow.

You can use this page to determine when the flow in your flow configuration is failing to complete. If you’re looking for details about flow executions within a specific time frame, you can search for the execution start date and time.

To view the messages for a specific flow execution, select the flow execution, and then choose View details in the upper right of the page.

The Flow executions messages page displays messages for each step and activity in the flow. It indicates when a step is scheduled, when a step starts, and when an activity starts. It displays the values of the flow parameters and the inputs that AWS IoT Things Graph passed to each step and activity in the flow. It also indicates when a failure occurred. You can use this page to troubleshoot flow execution failures.
Flow execution messages (7)

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Event type</th>
<th>Date created</th>
<th>Date updated</th>
<th>Payload</th>
</tr>
</thead>
</table>
| 4          | Activity failed| Sat, 11 May 2019 19:25:47 GMT | Invalid Date | ACTIVITY FAILED: [StepId:"Camera", env:"Request timeout", error:"com.amazonaws.iot.thinggraph.exceptions.TimeoutException: Request timeout!" at com.amazonaws.iot.thinggraph.commanager.SessionManager@18f11fe8]
| 3          | Activity started| Sat, 11 May 2019 19:25:47 GMT | Invalid Date | ACTIVITY_STARTED: [StepId:"Camera", shardId:"ba20c4e6-433-4997-81a0-eae8d8d4474", messageId:"1", flowInstanceId:"urn:iotdmus-west-2"]
| 2          | Activity scheduled| Sat, 11 May 2019 19:25:46 GMT | Invalid Date | ACTIVITY_SCHEDULED: [StepId:"Camera", input:"1", shardId:"ba20c4e6-433-4997-81a0-eae8d8d4474", messageId:"2", flowInstanceId:"urn:iotdmus-west-2"]
| 1          | Step started | Sat, 11 May 2019 19:25:46 GMT | Invalid Date | STEP_STARTED: [StepId:"Camera", shardId:"ba20c4e6-433-4997-81a0-eae8d8d4474", messageId:"1", flowInstanceId:"urn:iotdmus-west-2"]
| 0          | Execution started| Sat, 11 May 2019 19:25:46 GMT | Invalid Date | EXECUTION_STARTED: [flowParams:"1", screen1:"1", camera1:"1", templateId:"urn:iotdmus-west-2"]
Managing lifecycles for AWS IoT Things Graph entities, flows, systems, and deployments

This topic contains information about how to manage namespace versions and revisions of workflows (flows) and systems.

Namespace versioning protects your flows, systems, and deployment configurations from breaking when you update or create devices and device models (either your own or ones that get added when you sync with the public namespace). Changes that aren't backward compatible don't break existing deployments or even new deployments of flows that contain the earlier versions.

Additionally, this topic describes how to delete existing entities from your namespace, and deprecate and delete flows, systems, and deployment configurations.

Namespace management

A namespace is the container for entities created by using the AWS IoT Things Graph Data Model (TDM) language. These entities are associated with a specific account. The namespace is part of the URN for each entity.

In the following example, the namespace is everything before the device name (DL05PLCUnits).

\[urn:tdm:REGION/ACCOUNT_ID/default:deviceModel:DL05PLCUnits]\]

The following are the entities that are stored in your namespace:

- Properties
- States
- Events
- Actions
- Capabilities
- Mappings
- Devices
- Device models
- Services

Systems, Flows, and Deployments aren't stored in your namespace.

When you first upload an entity definition, AWS IoT Things Graph creates a namespace for you. This namespace is in sync with the current version of the public namespace. AWS IoT Things Graph provides a ValidateEntityDefinitions API that you can use to debug your TDM before you upload your entity definitions. After confirming that your TDM is valid, you upload the definitions by using either the UploadEntityDefinitions API or the AWS IoT Things Graph console.

AWS IoT Things Graph creates a new version of your namespace when you make a backward-incompatible change. The following are examples of actions that prompt the creation of a new namespace version:
Deleting systems, flows, and namespaces

- You update any existing entity (property, action, event, and so on) in your namespace.
- You sync your namespace with the public namespace.
- You deprecate the existing entities in your namespace and replace them with a new set of entities.

You can update existing entities by using either the AWS IoT Things Graph console or the `UploadEntityDefinitions` API. The `UploadEntityDefinitions` API has two optional Boolean parameters that enable you to perform the other two actions:

- `syncWithPublicNamespace`
- `deprecateExistingEntities`

After the `UploadEntityDefinitions` operation completes, there is a short delay before the entities are available in flows. When you're writing tests and scripted solutions, take this delay into account.

When you create a flow, by default AWS IoT Things Graph uses entities in the current version of your namespace. When you deploy a flow, by default AWS IoT Things Graph validates it against the current version of your namespace. This means that if you attempt to deploy a flow that contains entities that are incompatible with the current namespace, by default the deployment fails. A flow that contains deprecated entities also fails.

To specify an earlier namespace that contains entities that are compatible with a flow, use the optional `compatibleNamespaceVersion` parameter of the `CreateFlowTemplate` API. If you create the flow with this parameter, AWS IoT Things Graph validates your flow against the version value that you set with this parameter when you deploy the flow. The `CreateSystemTemplate` also has this optional parameter, so ensure that your system and flow are using the same namespace version when you deploy them.

The `UpdateFlowTemplate` and `UpdateSystemTemplate` APIs take the same optional `compatibleNamespaceVersion` parameter. This means you can use those operations to change the namespace version against which the flow is validated when it deploys.

Deleting systems, flows, and namespaces

To delete a flow in the AWS IoT Things Graph console, on the Flows page, select the flow to delete. Choose Delete.

When you create a flow in the AWS IoT Things Graph console, the console silently creates a system that contains the flow for you. When you delete the flow, the AWS IoT Things Graph console also silently deletes the system that contains the flow.

The `DeprecateFlowTemplate` and `DeprecateSystemTemplate` APIs enable you to mark a flow or system for deletion before you delete it. Deprecated systems and flows can't be deployed, but existing deployments that contain the system or flow continue to run. If you're not using the AWS IoT Things Graph console, you must deprecate a system or flow before you delete it. Delete a system before you delete the flows that it contains.
The `DeleteFlowTemplate` and `DeleteSystemTemplate` APIs enable you to delete a flow or system that is deprecated. After you delete a flow, any system that contains the flow no longer updates or deploys. After you delete a system, any deployment configuration that contains the system no longer updates or deploys. Existing deployments that contain the flow continue to run because they use a snapshot of the workflow that’s taken at the time of deployment. The same is true for deployments that contain deleted systems. You must delete all flows in a system before you delete the system.

To delete a namespace, use the `DeleteNamespace` API. Before you delete the namespace, you must delete all systems, flows, and flow configurations that use entities in the namespace. Existing flow deployments continue to work after you delete a namespace. AWS IoT Things Graph creates a snapshot of the flow, flow configuration, system, and entities for each deployment.

### Deleting flow configurations

Deleting a deployment configuration requires two steps.

1. **Undeploy the flow configuration.** This step removes the configuration from the AWS IoT Greengrass group or the cloud.
2. **Delete the flow configuration.** This step removes the deployment configuration from AWS IoT Things Graph.

The following procedure shows how to delete a deployment configuration by using the AWS IoT Things Graph console.

1. **Undeploy the flow configuration.**
   
   On the **Deployments** page, select the flow configuration to delete. In the upper right of the page, choose **Undeploy**. Enter **Undeploy**, and then choose **Undeploy** in the box that appears.

2. **Delete the deployment configuration.**
   
   On the **Deployments** page, select the flow configuration to delete. In the upper right of the page, choose **Delete**. Enter **delete**, and then choose **Delete** in the box that appears.

The `UndeploySystemInstance` and `DeleteSystemInstance` APIs also perform these two actions.
Security in AWS IoT Things Graph

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 Things Graph, 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 Things Graph. The following topics show you how to configure AWS IoT Things Graph 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 Things Graph resources.

**Topics**

- [AWS IoT Things Graph security](#) (p. 147)
- [Security in AWS IoT Greengrass deployments](#) (p. 149)
- [Security in AWS IoT Things Graph namespaces](#) (p. 150)
- [Data protection in AWS IoT Things Graph](#) (p. 151)
- [Identity and access management for AWS IoT Things Graph](#) (p. 153)
- [Monitoring AWS IoT Things Graph](#) (p. 165)
- [Compliance validation for AWS IoT Things Graph](#) (p. 174)
- [Resilience in AWS IoT Things Graph](#) (p. 174)
- [Infrastructure security in AWS IoT Things Graph](#) (p. 175)
- [Configuration and vulnerability analysis in AWS IoT Things Graph](#) (p. 175)
- [Security best practices for AWS IoT Things Graph](#) (p. 175)

**AWS IoT Things Graph security**

AWS IoT Things Graph uses X.509 certificates, managed subscriptions, AWS IoT policies, and IAM policies and roles to secure the applications that run on devices either in the cloud or in your local AWS IoT Greengrass environment. AWS IoT Things Graph uses the AWS IoT security features for cloud deployments and the AWS IoT Greengrass security features for AWS IoT Greengrass deployments.

For more information about AWS IoT security, see Security in AWS IoT in the AWS IoT Developer Guide.

For more information about AWS IoT Greengrass security, see AWS IoT Greengrass security in the AWS IoT Greengrass Developer Guide.
IAM role for flow execution in cloud deployments

Cloud deployments require you to use an IAM role to allow AWS IoT Things Graph to execute the flows in the deployments on your behalf. For more information about creating this role, see Prepare for cloud deployments.

The following diagram shows how AWS IoT Things Graph security fits within AWS IoT security.

IAM role for flow execution in AWS IoT Greengrass deployments

AWS IoT Greengrass deployments require you to use an IAM role to allow AWS IoT Things Graph to execute the flows in the deployments on your behalf. Your AWS IoT Greengrass service role must also have read and write permissions for Amazon S3. For more information about creating and setting up these roles, see Setting up your environment for AWS IoT Greengrass deployments.

The following diagram shows how AWS IoT Things Graph security fits within AWS IoT Greengrass security.
Security in AWS IoT Greengrass deployments

AWS IoT Things Graph uses the security features of AWS IoT Greengrass for AWS IoT Greengrass deployments. For more information, see AWS IoT Greengrass security in the AWS IoT Greengrass Developer Guide.

AWS IoT Things Graph users should have permissions to perform the following actions:

- s3:REST.GET.OBJECT
- s3:REST.PUT.OBJECT
- iot:CreateTopicRule
- iot:ListTopicRules
- iot:GetTopicRule
- iot:DescribeThing
- iot:ListThingPrincipals
- iot:UpdateThing
- iot:AddThingToThingGroup
- iot:DescribeThingGroup
- iot:CreateThingGroup
- iot:ListThingsInThingGroup
- iot:DeleteThingGroup
- iot:RemoveThingFromThingGroup
- greengrass:CreateGroupVersion
• greengrass:CreateDeployment
• greengrass:GetDeviceDefinitionVersion
• greengrass:CreateDeviceDefinitionVersion
• greengrass:GetDeviceDefinition
• greengrass:GetFunctionDefinitionVersion
• greengrass:CreateFunctionDefinitionVersion
• greengrass:GetFunctionDefinition
• greengrass:GetResourceDefinitionVersion
• greengrass:CreateResourceDefinitionVersion
• greengrass:GetResourceDefinition
• greengrass:GetSubscriptionDefinitionVersion
• greengrass:CreateSubscriptionDefinitionVersion
• greengrass:GetSubscriptionDefinition
• greengrass:ListGroups
• greengrass:GetGroupVersion
• greengrass:GetConnectorDefinitionVersion
• greengrass:CreateConnectorDefinition
• greengrass:GetConnectorDefinition
• iam:PassRole

**MQTT subscriptions**

AWS IoT Things Graph runs as an AWS Lambda function on the AWS IoT Greengrass core device. When new workflows are deployed to AWS IoT Greengrass, AWS IoT Things Graph creates entries in the subscription table for communication between the devices in the workflow and the AWS IoT Things Graph Lambda function.

AWS IoT Things Graph therefore creates subscriptions to all of the MQTT topics that the devices in your workflow use. The subscription table is used by the AWS IoT Greengrass core device for authorization of all communications. The AWS IoT Things Graph Lambda function mediates communications among the devices in the workflow.

**Security in AWS IoT Things Graph namespaces**

Every entity that you use in a workflow must belong to your namespace. For more information, see Namespaces.

Entities stored in an account’s namespace are available to all users in an account. All users in the account have read and write access to all entity definitions in the namespace.

Because you deploy AWS IoT Things Graph to AWS IoT Greengrass, you need to set up AWS IoT Greengrass security by following the steps in Configuring Greengrass security in the AWS IoT Greengrass Developer Guide.

When you deploy a workflow, you provide AWS IoT Things Graph with an IAM service role. This role should contain all of the policies required for AWS IoT Things Graph to publish and subscribe to all of the MQTT topics that are used in the workflow.

For more information about IAM roles, see Using IAM roles in the IAM User Guide.
Data protection in AWS IoT Things Graph

AWS IoT Things Graph conforms to the AWS shared responsibility model, which includes regulations and guidelines for data protection. AWS is responsible for protecting the global infrastructure that runs all the AWS services. AWS maintains control over data hosted on this infrastructure, including the security configuration controls for handling customer content and personal data. AWS customers and APN partners, acting either as data controllers or data processors, are responsible for any personal data that they put in the AWS Cloud.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM), so that 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.
- Set up API and user activity logging with AWS CloudTrail.
- 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 personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a Name field. This includes when you work with AWS IoT Things Graph or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into AWS IoT Things Graph or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't 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.

Data encryption

Data protection refers to protecting data while in-transit (as it travels to and from AWS IoT) and at rest (while it's stored on devices or by other AWS services). All data sent to AWS IoT is sent over an Transport Layer Security (TLS) connection using MQTT or HTTPS protocols, so it's secure by default in transit. AWS IoT devices collect data and then send 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.

Encryption at Rest

In cloud deployments of flows, AWS IoT Things Graph protects data at rest through server-side encryption. For more information, see Data encryption in AWS IoT in the AWS IoT Developer Guide.

In AWS IoT Greengrass deployments of flows, AWS IoT Things Graph uses the encryption features of AWS IoT Greengrass. For more information, see Encryption at Rest in the AWS IoT Greengrass Developer Guide.

Additionally, when AWS IoT Things Graph installs a flow to your AWS IoT Greengrass core, it first uploads a file that contains all of the flow's dependencies to an Amazon S3 bucket. AWS IoT Things Graph uploads this file by using the HTTPS protocol, so that it's secure in transit by default. It's your responsibility to secure the Amazon S3 bucket.
Encryption in transit

In cloud deployments of flows, AWS IoT Things Graph protects data in transit by using the Transport Layer Security (TLS) protocol. For more information, see Transport security in AWS IoT in the AWS IoT Developer Guide.

In AWS IoT Greengrass deployments of flows, AWS IoT Things Graph uses the data protection features of AWS IoT Greengrass. For more information, see Encryption in Transit in the AWS IoT Greengrass Developer Guide.

Additionally, when AWS IoT Things Graph installs a flow to your AWS IoT Greengrass core, it first uploads a file that contains all of the flow's dependencies to an Amazon S3 bucket. AWS IoT Things Graph uploads this file by using the HTTPS protocol, so it's secure in transit by default. It's your responsibility to secure the Amazon S3 bucket.

Key management

All connections to AWS IoT Things Graph are done using the Transport Layer Security (TLS) protocol, so client-side encryption keys aren't necessary for the initial TLS connection. Devices must authenticate using an X.509 certificate. AWS IoT can generate a certificate for you, in which case it generates a public/private key pair.

You can generate a certificate in the following ways:

- If you're using the AWS IoT console, you're prompted to download the certificate and keys.
- If you're using the create-keys-and-certificate AWS CLI command, the AWS CLI returns the certificate and keys.

**Important**
You're responsible for copying the certificate and private key to your device and keeping it safe.

Internetwork traffic privacy

In cloud deployments, AWS IoT Things Graph doesn't support connections with on-premises applications, or between AWS accounts and Regions.

In AWS IoT Greengrass deployments, AWS IoT Greengrass doesn't encrypt data in transit locally between components on the AWS IoT Greengrass core. This includes communication from group members (such as AWS Lambda functions and connectors), the AWS IoT Greengrass Core SDK, and stream manager.

Data retention

The following characteristics of AWS IoT Things Graph relate to data retention:

- When you use an API to delete an AWS IoT Things Graph resource, the service deletes that resource immediately.
- AWS IoT Things Graph retains some flow execution log data until you delete your AWS account. When you delete your account, AWS IoT Things Graph deletes all remaining log data.
- When you create a new version of a namespace, AWS IoT Things Graph takes a snapshot of the earlier version. These earlier versions of the namespace are deleted when you delete the namespace.
- When you deploy a flow to AWS IoT Greengrass, AWS IoT Things Graph creates a deployment artifact that contains the dependency closure of the flow configuration, and saves it to an Amazon S3 bucket that you specify. After the deployment is complete, you can delete this artifact.
Identity and access management for AWS IoT Things Graph

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 Things Graph resources. IAM is an AWS service that you can use with no additional charge.

Topics
- Audience (p. 153)
- Authenticating with identities (p. 153)
- Managing access using policies (p. 155)
- How AWS IoT Things Graph works with IAM (p. 156)
- AWS IoT Things Graph identity-based policy examples (p. 160)
- Troubleshooting AWS IoT Things Graph identity and access (p. 163)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in AWS IoT Things Graph.

Service user – If you use the AWS IoT Things Graph service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS IoT Things Graph features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS IoT Things Graph, see Troubleshooting AWS IoT Things Graph identity and access (p. 163).

Service administrator – If you're in charge of AWS IoT Things Graph resources at your company, you probably have full access to AWS IoT Things Graph. It's your job to determine which AWS IoT Things Graph features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS IoT Things Graph, see How AWS IoT Things Graph works with IAM (p. 156).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS IoT Things Graph. To view example AWS IoT Things Graph identity-based policies that you can use in IAM, see AWS IoT Things Graph identity-based policy examples (p. 160).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see The IAM Console and Sign-in Page in the IAM User Guide.

You must be authenticated (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication, or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles.

When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email or your IAM user name. You can access AWS programmatically using your root user or IAM user access keys. AWS
provides SDK and command line tools to cryptographically sign your request using your credentials. If you don’t use AWS tools, you must sign the request yourself. Do this using Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the AWS General Reference.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using Multi-Factor Authentication (MFA) in AWS in the IAM User Guide.

AWS Account Root User

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing Access Keys for IAM Users in the IAM User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An IAM group is an identity that specifies a collection of IAM users. You can’t sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named IAMAdmins and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to Create an IAM User (Instead of a Role) in the IAM User Guide.

IAM roles

An IAM role is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM Roles in the IAM User Guide.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the IAM User Guide.
Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines its permissions. AWS evaluates these policies when an entity (root user, IAM user, or IAM role) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON Policies in the IAM User Guide.

An IAM administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the iam:GetRole action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, role, or group. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM Policies in the IAM User Guide.

Identity-based policies can be further categorized as inline policies or managed policies. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that
you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing 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 such as an Amazon S3 bucket. Service administrators can use these policies to define what actions a specified principal (account member, user, or role) can perform on that resource and under what conditions. Resource-based policies are inline policies. There are no managed resource-based policies.

Access control lists (ACLs)

Access control lists (ACLs) are a type of policy that controls 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, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity’s identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions Boundaries for IAM Entities in the IAM User Guide.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs Work in the AWS Organizations User Guide.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. 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 Things Graph works with IAM

Before you use IAM to manage access to AWS IoT Things Graph, you should understand what IAM features are available to use with AWS IoT Things Graph. To get a high-level view of how AWS IoT Things
Graph and other AWS services work with IAM, see AWS services that work with IAM in the IAM User Guide.

Topics
- AWS IoT Things Graph identity-based policies (p. 157)
- AWS IoT Things Graph resource-based policies (p. 159)
- Access control lists (p. 159)
- Authorization based on AWS IoT Things Graph tags (p. 159)
- AWS IoT Things Graph IAM roles (p. 160)

AWS IoT Things Graph 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 Things Graph 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

The Action element of an IAM identity-based policy describes the specific action or actions that will be allowed or denied by the policy. Policy actions usually have the same name as the associated AWS API operation. The action is used in a policy to grant permissions to perform the associated operation.

The AWS IoT Things Graph control plane uses IAM policies to define what a user can do. You can create policies that assign permissions for all actions in the AWS IoT Things Graph API Reference.

For example, the following policy assigns permission for a user to use the CreateSystemInstance API.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iotthingsgraph:CreateSystemInstance"
            ],
            "Resource": [ "*" ]
        }
    ]
}
```

The following policy gives permission to perform all actions.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iotthingsgraph:*"
            ],
            "Resource": [ "*" ]
        }
    ]
}
```
Policy actions in AWS IoT Things Graph use the following prefix before the action: `iotthingsgraph:`. For example, to grant someone permission to upload `iotthingsgraph` entities with the `AWS IoT Things Graph` `UploadEntityDefinitions` API operation, you include the `iotthingsgraph:UploadEntityDefinitions` action in the policy. Policy statements must include an `Action` or `NotAction` element. AWS IoT Things Graph 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": [  
  "iotthingsgraph:action1",
  "iotthingsgraph:action2"
]
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Create`, include the following action.

```
"Action": "iotthingsgraph:Create*"
```

To see a list of AWS IoT Things Graph actions, see Actions Defined by AWS IoT Things Graph in the IAM User Guide.

**Resources**

The `Resource` element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. You specify a resource using an ARN or using the wildcard (*) to indicate that the statement applies to all resources.

The AWS IoT Things Graph workflow (flow) instance resource has the following ARN.

```
arn:${Partition}:iotthingsgraph:${Region}:${Account}:workflow/${WorkflowName}
```

For more information about the format of ARNs, see Amazon Resource Names (ARNs) in the AWS General Reference.

For example, to specify the `SecurityFlow` flow in your statement, use the following ARN.

```
```

To specify all flows that belong to a specific account, use the wildcard (*).

```
"Resource": "arn:aws:iotthingsgraph:us-east-1:123456789012:workflow/*"
```

Some AWS IoT Things Graph actions, such as those for creating resources, can't be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
```

158
To see a list of AWS IoT Things Graph resource types and their ARNs, see Resources Defined by AWS IoT Things Graph in the IAM User Guide. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS IoT Things Graph.

**Condition keys**

The Condition element (or Condition block) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can build conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM Policy Elements: Variables and Tags in the IAM User Guide.

AWS IoT Things Graph 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.

To see a list of AWS IoT Things Graph condition keys, see Condition Keys for AWS IoT Things Graph in the IAM User Guide. To learn with which actions and resources you can use a condition key, see Actions Defined by AWS IoT Things Graph.

**Examples**

To view examples of AWS IoT Things Graph identity-based policies, see AWS IoT Things Graph identity-based policy examples (p. 160).

**AWS IoT Things Graph resource-based policies**

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on the AWS IoT Things Graph resource and under what conditions. AWS IoT Things Graph doesn't support IAM resource-based policies.

**Access control lists**

Access control lists (ACLs) are lists of grantees that you can attach to resources. AWS IoT Things Graph doesn't support ACLs.

**Authorization based on AWS IoT Things Graph tags**

You can attach tags to AWS IoT Things Graph resources or pass tags in a request to AWS IoT Things Graph. To control access based on tags, you provide tag information in the condition element of a policy using the iotthingsgraph:ResourceTag/key-name, aws:RequestTag/key-name, or aws:TagKeys condition keys. For more information about tagging AWS IoT Things Graph resources, see Tagging your AWS IoT Things Graph 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 Things Graph flow configurations based on tags (p. 162).
AWS IoT Things Graph IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT Things Graph

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 Things Graph 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. AWS IoT Things Graph doesn’t support 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 Things Graph supports service roles.

AWS IoT Things Graph identity-based policy examples

By default, IAM users and roles don’t have permission to create or modify AWS IoT Things Graph 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 IAM 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 (p. 160)
- Using the AWS IoT Things Graph console (p. 161)
- Allow users to view their own permissions (p. 161)
- Getting an AWS IoT Things Graph flow (p. 162)
- Viewing AWS IoT Things Graph flow configurations based on tags (p. 162)

Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete AWS IoT Things Graph 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 Using AWS Managed Policies – To start using AWS IoT Things Graph quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get Started Using Permissions With AWS Managed Policies in the IAM User Guide.
• **Grant Least Privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see **Grant Least Privilege** in the [IAM User Guide](https://docs.aws.amazon.com/IAM/latest/UserGuide/IAM_user_guide.html).


• **Use Policy Conditions for Extra Security** – To the extent that it’s practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see [IAM JSON Policy Elements: Condition](https://docs.aws.amazon.com/IAM/latest/UserGuide/policy-conditions.html) in the [IAM User Guide](https://docs.aws.amazon.com/IAM/latest/UserGuide/IAM_user_guide.html).

**Using the AWS IoT Things Graph console**

To access the AWS IoT Things Graph console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS IoT Things Graph 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 (IAM users or roles) with that policy.

To ensure that those entities can still use the AWS IoT Things Graph console, make sure that they have permissions to perform the following actions:

- `s3:REST.GET.OBJECT`
- `s3:REST.PUT.OBJECT`
- `iot:CreateTopicRule`
- `iot:ListTopicRules`
- `iot:GetTopicRule`
- `iot:DescribeThing`
- `iot:ListThingPrincipals`
- `iot:UpdateThing`
- `iot:AddThingToThingGroup`
- `iot:DescribeThingGroup`
- `iot:CreateThingGroup`
- `iot:ListThingsInThingGroup`
- `iot:DeleteThingGroup`
- `iot:RemoveThingFromThingGroup`

For a list of required permissions for users who create and deploy flows to AWS IoT Greengrass, see [Security in AWS IoT Greengrass deployments](https://docs.aws.amazon.com/IoT/latest/DeveloperGuide/greengrass-security.html) (p. 149).

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.
Identity-based policy examples

```json
{
   "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}"],
         "Resource": ["*"]
      },
      {
         "Sid": "NavigateInConsole",
         "Effect": "Allow",
         "Action": [
            "iam:GetGroupPolicy",
            "iam:GetPolicyVersion",
            "iam:GetPolicy",
            "iam:ListAttachedGroupPolicies",
            "iam:ListGroupPolicies",
            "iam:ListPolicyVersions",
            "iam:ListPolicies",
            "iam:ListUsers"
         ],
         "Resource": ["*"]
      }
   ]
}
```

Getting an AWS IoT Things Graph flow

In this example, you want to grant an IAM user in your AWS account access to your AWS IoT Things Graph flows.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "ViewFlows",
         "Effect": "Allow",
         "Action": "iotthingsgraph:GetFlowTemplate",
         "Resource": ["*"]
      }
   ]
}
```

Viewing AWS IoT Things Graph flow configurations based on tags

You can use conditions in your identity-based policy to control access to AWS IoT Things Graph resources based on tags. This example shows how you might create a policy that allows viewing a Deployment (flow configuration). However, permission is granted only if the Deployment 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.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "FilterFlows",
         "Effect": "Allow",
         "Action": "iotthingsgraph:GetFlowTemplate",
         "Resource": ["*"]
      }
   ]
}
```
You can attach this policy to the IAM users in your account. If a user named richard-roe attempts to view an AWS IoT Things Graph flow configuration, the flow configuration 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.

**Troubleshooting AWS IoT Things Graph identity and access**

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT Things Graph and IAM.

**Topics**

- My flow executions fail with an access denied error (p. 163)
- I am not authorized to perform an action in AWS IoT Things Graph (p. 164)
- I am not authorized to perform iam:PassRole (p. 164)
- I want to view my access keys (p. 164)
- I'm an administrator and want to allow others to access AWS IoT Things Graph (p. 165)
- I want to allow people outside of my AWS account to access my AWS IoT Things Graph resources (p. 165)

**My flow executions fail with an access denied error**

A flow execution might fail with this error for one of the following reasons:

- The IAM role that you specified for flow execution in a cloud deployment doesn't have a trust relationship with AWS IoT Things Graph. AWS IoT Things Graph can't assume this role.
- The IAM role that you specified for flow execution in a cloud deployment or the IAM role for AWS IoT Greengrass that you specified for your AWS IoT Greengrass group doesn't have permission to use the AWS IoT DescribeEndpoint API. AWS IoT Things Graph can't find your endpoint and send messages to it.
- The IAM role that you specified for flow execution in a cloud deployment doesn't have permission to use a resource or API from another service. For example, if this role doesn't have permission to use the AWS IoT Publish API, AWS IoT Things Graph can't send messages to MQTT topics.
I am not authorized to perform an action in AWS IoT Things Graph

If the AWS Management Console tells you that you’re not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the mateojackson IAM user tries to use the console to view details about a flow but doesn’t have iotthingsgraph:GetFlowTemplate permissions.

```plaintext
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: iotthingsgraph:GetFlowTemplate on resource: my-example-flow
```

In this case, Mateo asks his administrator to update his policies to allow him to access the my-example-flow resource using the iotthingsgraph:GetFlowTemplate 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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to AWS IoT Things Graph.

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 Things Graph. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.

```plaintext
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary asks her administrator to update her policies to allow her to perform the iam:PassRole action.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can’t view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, AKIAIOSFODNN7EXAMPLE) and a secret access key (for example, wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

**Important**

Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing Access Keys in the IAM User Guide.
I'm an administrator and want to allow others to access AWS IoT Things Graph

To allow others to access AWS IoT Things Graph, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in AWS IoT Things Graph.

To get started right away, see Creating Your First IAM Delegated User and Group in the IAM User Guide.

I want to allow people outside of my AWS account to access my AWS IoT Things Graph 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 Things Graph supports these features, see How AWS IoT Things Graph works with IAM (p. 156).
- 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 How IAM Roles Differ from Resource-based Policies in the IAM User Guide.

Monitoring AWS IoT Things Graph

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT Things Graph and your other AWS solutions. AWS provides the following monitoring tools to watch AWS IoT Things Graph, report when something is wrong, and take automatic actions when appropriate.

- **Amazon CloudWatch** monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the Amazon CloudWatch User Guide.
- **Amazon CloudWatch Events** delivers a near real-time stream of system events that describe changes in AWS resources. CloudWatch Events enables automated event-driven computing, because you can write rules that watch for certain events and trigger automated actions in other AWS services when these events happen. For more information, see the Amazon CloudWatch Events User Guide.
- **Amazon CloudWatch Logs** lets you monitor, store, and access your log files from Amazon EC2 instances, CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the Amazon CloudWatch Logs User Guide.
- **AWS CloudTrail** captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users
and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Topics

- Monitoring AWS IoT Things Graph with Amazon CloudWatch (p. 166)
- Logging AWS IoT Things Graph API calls with AWS CloudTrail (p. 167)

Monitoring AWS IoT Things Graph with Amazon CloudWatch

You can monitor AWS IoT Things Graph using CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the Amazon CloudWatch User Guide.

CloudWatch collects data only for cloud deployments of flow configurations. CloudWatch isn't available for AWS IoT Greengrass deployments.

To enable data collection using the console

1. Sign in to the AWS IoT Things Graph console.
2. On the Configure target page, select Enable metrics. This page appears after you enter a name for a flow.

Select the radio button next to the Enable metrics option.

To enable data collection using the CLI

- Create a MetricsConfiguration object and pass it to the CreateSystemInstance command.

The MetricsConfiguration object contains the following values.

- cloudMetricEnabled: A Boolean that specifies whether cloud metrics are collected.
- metricRuleRoleArn: The ARN of an IAM role used to collect cloud metrics. This value is not required.

Example

The following AWS CLI example enables data collection for a new flow configuration.

```
aws iotthingsgraph create-system-instance \
  --definition language=GRAPHQL,text="MySystemInstanceDefinition" \
  --target CLOUD \
  --flow-actions-role-arn myRoleARN \
  --metrics-configuration cloudMetricEnabled=true
```

The following tables list the metrics and dimensions for AWS IoT Things Graph.

### AWS IoT Things Graph metrics

AWS IoT Things Graph sends the following metrics to CloudWatch every minute.
## AWS IoT Things Graph User Guide

### Logging AWS IoT Things Graph API calls with AWS CloudTrail

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowExecutionTime</td>
<td>The amount of time a flow execution takes to complete.</td>
</tr>
<tr>
<td>FlowExecutionsAborted</td>
<td>The number of flow executions that are aborted.</td>
</tr>
<tr>
<td>FlowExecutionsFailed</td>
<td>The number of flow executions that failed.</td>
</tr>
<tr>
<td>FlowExecutionsStarted</td>
<td>The number of flow executions that started.</td>
</tr>
<tr>
<td>FlowExecutionsSucceeded</td>
<td>The number of flow executions that succeeded.</td>
</tr>
<tr>
<td>FlowStepExecutionTime</td>
<td>The amount of time a flow execution step takes to complete.</td>
</tr>
<tr>
<td>FlowStepExecutionsFailed</td>
<td>The number of flow execution steps that failed.</td>
</tr>
<tr>
<td>FlowStepExecutionsStarted</td>
<td>The number of flow execution steps that started.</td>
</tr>
<tr>
<td>FlowStepExecutionsSucceeded</td>
<td>The number of flow execution steps that succeeded.</td>
</tr>
<tr>
<td>FlowStepLambdaExecutionsFailed</td>
<td>The number of flow execution Lambda function steps that failed.</td>
</tr>
<tr>
<td>FlowStepLambdaExecutionsStarted</td>
<td>The number of flow execution Lambda function steps that started.</td>
</tr>
<tr>
<td>FlowStepLambdaExecutionsSucceeded</td>
<td>The number of flow execution Lambda function steps that succeeded.</td>
</tr>
</tbody>
</table>

### Dimensions for AWS IoT Things Graph metrics

AWS IoT Things Graph metrics use the AWS IoT Things Graph namespace and provide metrics for the following dimensions.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowTemplateId</td>
<td>The ID of the flow template that the flow execution belongs to.</td>
</tr>
<tr>
<td>LambdaArn</td>
<td>The ARN of the Lambda function flow execution step.</td>
</tr>
<tr>
<td>StepName</td>
<td>The name of the flow execution step.</td>
</tr>
<tr>
<td>SystemTemplateId</td>
<td>The ID of the system template that the flow execution belongs to.</td>
</tr>
</tbody>
</table>

### Logging AWS IoT Things Graph API calls with AWS CloudTrail

AWS IoT Things Graph is integrated with AWS CloudTrail. The CloudTrail service provides a record of actions that a user, role, or an AWS service takes in AWS IoT Things Graph. CloudTrail captures all API calls for AWS IoT Things Graph as events. Captured calls include those from the AWS IoT Things Graph console and code calls to the AWS IoT Things Graph API operations.
If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS IoT Things Graph. 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 that CloudTrail collects, you can determine the request that was made to AWS IoT Things Graph, the IP address from which the request was made, who made the request and when, and more details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS IoT Things Graph information in CloudTrail

AWS CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT Things Graph, that activity is recorded in a CloudTrail event 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 Things Graph, create a trail. A trail enables CloudTrail to deliver log files to an Amazon Simple Storage Service (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 Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify.

You can also configure other AWS services to further analyze and act on the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple Regions
- Receiving CloudTrail log files from multiple accounts

CloudTrail logs all AWS IoT Things Graph actions. They’re documented in the AWS IoT Things Graph API Reference. For example, calls to the UploadEntityDefinitions, CreateSystemInstance and CreateFlowTemplate actions generate entries in the CloudTrail log files.

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 AWS Identity and Access Management (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.

Understanding AWS IoT Things Graph 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 aren’t an ordered stack trace of the public API calls, so they don’t appear in any specific order.
Example

The following CloudTrail log entry shows information about the `UploadEntityDefinitions` action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "principal-id",
    "arn": "arn:aws:iam::123456789012:user/TodRunner",
    "accountId": "123456789012",
    "accessKeyId": "access-key-id",
    "userName": "TodRunner"
  },
  "eventTime": "2019-02-22T22:56:35Z",
  "eventSource": "iotthingsgraph.amazonaws.com",
  "eventName": "UploadEntityDefinitions",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "127.0.0.1",
  "userAgent": "aws-internal/3 aws-sdk-java/1.11.498 Mac_OS_X/10.12.6 Java_HotSpot(TM)_64-Bit_Server_VM/25.181-b13 java/1.8.0_181",
  "requestParameters": {
    "document": {
      "language": "GRAPHQL",
      lastClickedImage : Uri @property(id: \"urn:tdm:us-east-1/123456789012/default:Property:CameraStateAcceptanceSimpleFlow\")\n      }\n      type CameraState_V2 @stateType(id: \"urn:tdm:us-east-1/123456789012/default:State:CameraState_V2AcceptanceSimpleFlow\")
      lastClickedImage : Uri @property(id: \"urn:tdm:us-east-1/123456789012/default:Property:CameraState_V2AcceptanceSimpleFlow\")\n      }\n      type ScreenState @stateType(id: \"urn:tdm:us-east-1/123456789012/default:State:ScreenStateAcceptanceSimpleFlow\")
      currentDisplayImage : Uri @property(id: \"urn:tdm:us-east-1/123456789012/default:Property:ScreenStateAcceptanceSimpleFlow\")\n      }\n      type MotionSensorState @stateType(id: \"urn:tdm:us-east-1/123456789012/default:State:MotionSensorStateAcceptanceSimpleFlow\")
      isMotionDetected : MotionDetected @property(id: \"urn:tdm:us-east-1/123456789012/default:Property:MotionDetectedAcceptanceSimpleFlow\")
    }
  }
}
```
Logging AWS IoT Things Graph API calls with AWS CloudTrail

```typescript
// AWS IoT Things Graph User Guide

import { LightState } from '@stateType(id: "urn:tdm:us-east-1/123456789012/default:State:LightStateAcceptanceSimpleFlow")
{
  OnOff : Boolean

  type BoundingBoxState
  @stateType(id:'urn:tdm:us-east-1/123456789012/default:State:SimpleFlow')
  {
    Height: Float64
    @property(id: 'urn:tdm:aws:Property:Float64'),
    Width: Float64
    @property(id: 'urn:tdm:aws:Property:Float64'))

  type FaceDetailsState
  @stateType(id:'urn:tdm:us-east-1/123456789012/default:State:FaceDetailsStateAcceptanceSimpleFlow')
  {
    BoundingBox:
    BoundingBoxProperty
    @property(id:'urn:tdm:us-east-1/123456789012/default:Property:BoundingBoxPropertyAcceptanceSimpleFlow'),
    Confidence: Float64
    @property(id:'urn:tdm:aws:Property:Float64'),
    Landmarks: [LandmarksProperty]
    @property(id:'urn:tdm:us-east-1/123456789012/default:Property:LandmarksPropertyAcceptanceSimpleFlow'),
    Pose: PoseProperty
    @property(id:'urn:tdm:aws:Property:PoseProperty'),
    Quality:
    QualityProperty
    @property(id:'urn:tdm:aws:Property:QualityProperty'),
    QualityState
  }@stateType(id:'urn:tdm:us-east-1/123456789012/default:State:QualityStateAcceptanceSimpleFlow')
  {
    Brightness:
    Float64
    @property(id:'urn:tdm:aws:Property:Float64'),
    Sharpness: Float64
    @property(id:'urn:tdm:aws:Property:Float64')
  }

  type ResponseState
  @stateType(id:'urn:tdm:us-east-1/123456789012/default:State:ResponseStateAcceptanceSimpleFlow')
  {
    FaceDetails:
    [FaceDetailsProperty]
    @property(id:'urn:tdm:us-east-1/123456789012/default:Property:FaceDetailsPropertyAcceptanceSimpleFlow'),
    OrientationCorrection:
    String
    @property(id:'urn:tdm:aws:Property:String'),
    X: Float64
    @property(id:'urn:tdm:aws:Property:Float64'),
    Y: Float64
    @property(id:'urn:tdm:aws:Property:Float64')
  }

  type CreateQueue
  @actionType(id: "urn:tdm:us-east-1/123456789012/default:Action:CreateQueueAcceptanceSimpleFlow")
  {
    queueName: String
    @property(id: 'urn:tdm:aws:Property:String')
  }

  type SendMessage
  @actionType(id: "urn:tdm:us-east-1/123456789012/default:Action:SendMessageAcceptanceSimpleFlow")
  {
    message: String
    @property(id: 'urn:tdm:aws:Property:String')
  }

  type DetectFaces
  @actionType(id: "urn:tdm:us-east-1/123456789012/default:Action:DetectFacesAcceptanceSimpleFlow")
  {
    bucketName: String
    @property(id: 'urn:tdm:aws:Property:Bucket'),
    itemName: String
    @property(id: 'urn:tdm:aws:Property:Item'),
    return: ResponseProperty
    @property(id: 'urn:tdm:us-east-1/123456789012/default:Property:ResponsePropertyAcceptanceSimpleFlow')
  }

  type MotionDetectedEvent
  @eventType(id: "urn:tdm:us-east-1/123456789012/default:Event:MotionDetectedEventAcceptanceSimpleFlow",
  payload: "urn:tdm:us-east-1/123456789012/default:Property:MotionDetectedAcceptanceSimpleFlow")
  {
    ignore: void
  }

  type CameraCap
  @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:CameraCapAcceptanceSimpleFlow")
  {
    STATE: CameraState
    @state(id: "urn:tdm:us-east-1/123456789012/default:State:CameraStateAcceptanceSimpleFlow")
  }
```

170
capture: Capture @action(id: "urn:tdm:us-east-1/123456789012/default:Action:CaptureAcceptanceSimpleFlow")
(default:CameraCap_V2 @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:CameraCap_V2AcceptanceSimpleFlow")
)
STATE: CameraState_V2 @state(id: "urn:tdm:us-east-1/123456789012/default:State:CameraState_V2AcceptanceSimpleFlow")
capture: Capture @action(id: "urn:tdm:us-east-1/123456789012/default:Action:CaptureAcceptanceSimpleFlow")
(default:ScreenCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:ScreenCapAcceptanceSimpleFlow")
)
STATE: ScreenState @state(id: "urn:tdm:us-east-1/123456789012/default:State:ScreenStateAcceptanceSimpleFlow")
display: Display @action(id: "urn:tdm:us-east-1/123456789012/default:Action:DisplayAcceptanceSimpleFlow")

DEFAULT: CameraCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:CameraCapAcceptanceSimpleFlow")

DEFAULT: ScreenCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:ScreenCapAcceptanceSimpleFlow")

DEFAULT: MotionSensorCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:MotionSensorCapAcceptanceSimpleFlow")

DEFAULT: SqsCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:capability:SqsCapAcceptanceSimpleFlow")

DEFAULT: RekognitionCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:capability:RekognitionCapAcceptanceSimpleFlow")

DEFAULT: LightCap @capabilityType(id: "urn:tdm:us-east-1/123456789012/default:Capability:LightCapAcceptanceSimpleFlow")

DEFAULT: Camera @deviceModel(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:CameraAcceptanceSimpleFlow")

DEFAULT: Camera_V2 @deviceModel(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:Camera_V2AcceptanceSimpleFlow")

DEFAULT: Screen @deviceModel(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:ScreenAcceptanceSimpleFlow")

DEFAULT: MotionSensor @deviceModel(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:MotionSensorAcceptanceSimpleFlow")

DEFAULT: Light @deviceModel(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:LightAcceptanceSimpleFlow")

DEFAULT: PhyCamera @device(id: "urn:tdm:us-east-1/123456789012/default:Device:PhyCameraAcceptanceSimpleFlow")

state: LightState @state(id: "urn:tdm:us-east-1/123456789012/default:State:LightStateAcceptanceSimpleFlow")
switch: Switch @action(id: "urn:tdm:us-east-1/123456789012/default:Action:SwitchAcceptanceSimpleFlow")

CameraCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:CameraCapAcceptanceSimpleFlow")

ScreenCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:ScreenCapAcceptanceSimpleFlow")

MotionSensorCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:MotionSensorCapAcceptanceSimpleFlow")

LightCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:LightCapAcceptanceSimpleFlow")

Camera_V2(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:Camera_V2AcceptanceSimpleFlow")

Screen_V2(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:Screen_V2AcceptanceSimpleFlow")

MotionSensor_V2(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:MotionSensor_V2AcceptanceSimpleFlow")

Light_V2(id: "urn:tdm:us-east-1/123456789012/default:DeviceModel:Light_V2AcceptanceSimpleFlow")

PhyCamera(id: "urn:tdm:us-east-1/123456789012/default:Device:PhyCameraAcceptanceSimpleFlow")

MQTT {CameraCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:CameraCapAcceptanceSimpleFlow")

MQTT {ScreenCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:ScreenCapAcceptanceSimpleFlow")

MQTT {MotionSensorCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:MotionSensorCapAcceptanceSimpleFlow")

MQTT {LightCap(id: "urn:tdm:us-east-1/123456789012/default:Capability:LightCapAcceptanceSimpleFlow")

MQTT {PhyCamera(id: "urn:tdm:us-east-1/123456789012/default:Device:PhyCameraAcceptanceSimpleFlow")

MQTT {deviceModel: "urn:tdm:us-east-1/123456789012/default:DeviceModel:CameraAcceptanceSimpleFlow"}}

lastClickedImage(name: "lastImage", property: "urn:tdm:us-east-1/123456789012/default:Property:UriAcceptanceSimpleFlow")
Action(name: "capture")

responsePayload(property: 171)
\"urn:tdm:us-east-1/123456789012/default:Property:UriAcceptanceSimpleFlow\")

{
    MQTT {
        CameraCap_V2(id: \"urn:tdm:us-east-1/123456789012/default:Capability:CameraCap_V2AcceptanceSimpleFlow\") {
            state {
            }
            Action(name: \"capture\") {
                Publish {
                    Request(topic: \"camera_v2/capture\") {
                        params
                        Request(topic: \"camera_v2/capture/finished\") {
                        }
                    }
                }
            }
        }
    }

{
    MQTT {
        ScreenCap(id: \"urn:tdm:us-east-1/123456789012/default:Capability:ScreenCapAcceptanceSimpleFlow\") {
            state {
            }
            Action(name: \"display\") {
                params {
                }
                Payload {
                    Request(topic: \"screen/display\") {
                        params {
                        value: \"\${url.value}\"
                        }
                    }
                }
            }
        }
    }

query PhyMotionSensor @device(id: \"urn:tdm:us-east-1/123456789012/default:Device:PhyMotionSensorAcceptanceSimpleFlow\", deviceModel: \"urn:tdm:us-east-1/123456789012/default:DeviceModel:MotionSensorAcceptanceSimpleFlow\")
{
    MQTT {
        MotionSensorCap(id: \"urn:tdm:us-east-1/123456789012/default:Capability:MotionSensorCapAcceptanceSimpleFlow\") {
            state {
                isMotionDetected(name: \"\", property: \"urn:tdm:us-east-1/123456789012/default:Property:MotionDetectedAcceptanceSimpleFlow\")
            }
            Event(name: \"motionDetected\") {
                Subscribe(topic: \"motionsensor/motion\") {
                    responsePayload(property: \"urn:tdm:us-east-1/123456789012/default:Property:MotionDetectedAcceptanceSimpleFlow\")
                }
            }
        }
    }

{
    MODBUS(serverId: \"1\") {
        LightCap(id: \"urn:tdm:us-east-1/123456789012/default:Capability:LightCapAcceptanceSimpleFlow\") {
            state {
                OnOff(name: \"ON_OFF\", property: \"urn:tdm:aws:Property:Boolean\")
            }
            Action(name: \"switch\") {
                params {
                    param(name: \"input\", property: \"urn:tdm:aws:Property:Boolean\")
                    WriteSingleCoil{
                        Request(Address: 1) {
                            params {
                                property: \"urn:tdm:aws:Property:Boolean\" value: \"\${input.value ?? false}\"
                            }
                        }
                    }
                }
            }
        }
    }

query MLLambdas @service(id: \"urn:tdm:us-east-1/123456789012/default:Service:MLLambdasAcceptanceSimpleFlow\")
{
    AwsLambda {
        MLLambdasCap(id: \"urn:tdm:us-east-1/123456789012/default:Capability:MLLambdasCapAcceptanceSimpleFlow\") {
            Action(name: \"matchImageLocal\") {
                invokeGreengrassLambda {
                        params {
                        }
                    }
                }
            }
        }
    }
}
AWS IoT Things Graph User Guide
Logging AWS IoT Things Graph
API calls with AWS CloudTrail

```
Action(name: "matchImageGlobal") {
  params {
    param(name: "url", property: "urn:tdm:us-east-1/123456789012/default:property:UriAcceptanceSimpleFlow")
  }
  Response {
    responsePayload(property: 'urn:tdm:aws:Property:Boolean')
  }
}

InvokeCloudLambda {
  Request(
    arn: "arn:aws:lambda:us-east-1:123456789012:function:rekognition:1") {
  params {
    param(name: "imageUri", property: "urn:tdm:us-east-1/123456789012/default:Property:UriAcceptanceSimpleFlow")
  }
  Response {
    responsePayload(property: 'urn:tdm:aws:Property:Boolean')
  }
}

Action(name: "createQueue") {
  params {
    param(name: "queueName", property: "urn:tdm:aws:Property:String")
  }
  HttpGet {
    Request(url: "https://sqs.us-east-1.amazonaws.com", auth: "SigV4", awsServiceName: "sqs") {
      headerParams {
        param(name: "Content-Type", property: 'urn:tdm:aws:Property:String', value: "application/json")
      }
      queryParams {
        param(name: "QueueName", property: "urn:tdm:aws:Property:String", value: "${queueName.value}")
      }
    }
  }
}

Action(name: "sendMessage") {
  params {
    param(name: "message", property: "urn:tdm:aws:Property:String")
  }
  HttpPost {
    Request(url: "https://rekognition.us-east-1.amazonaws.com", auth: "SigV4", awsServiceName: "rekognition") {
      headerParams {
        param(name: "Content-Type", property: 'urn:tdm:aws:Property:String', value: "application/x-amz-json-1.1")
      }
      queryParams {
      }
    }
  }

tools
```
Compliance validation for AWS IoT Things Graph

Third-party auditors assess the security and compliance of AWS IoT Things Graph as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. 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 IoT Things Graph is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- **Security and Compliance Quick Start Guides** – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- **Architecting for HIPAA Security and Compliance Whitepaper** – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- **AWS compliance resources** – This collection of workbooks and guides might apply to your industry and location.
- **Evaluating resources with rules in the AWS Config Developer Guide** – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- **AWS Security Hub** – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in AWS IoT Things Graph

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 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.

In addition to the AWS global infrastructure, AWS IoT Things Graph offers several features to help support your data resiliency and backup needs.

Infrastructure security in AWS IoT Things Graph

As a managed service, AWS IoT Things Graph 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 Things Graph through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend 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.

Additionally, 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.

Configuration and vulnerability analysis in AWS IoT Things Graph

This section describes the actions that you must take to manage your AWS IoT Things Graph entities, deployments, and namespace.

For information about vulnerability management and analysis in AWS IoT, see Vulnerability analysis and management in AWS IoT in the AWS IoT Developer Guide.

AWS IoT Greengrass environments

It's your responsibility to configure and maintain your AWS IoT Greengrass environment. This responsibility includes upgrading to the latest version of the AWS IoT Greengrass software. For more information, see Configure the AWS IoT Greengrass core in the AWS IoT Greengrass Developer Guide.

Lifecycle management

It's your responsibility to manage the lifecycles of your AWS IoT Things Graph flows, flow configurations, deployments, systems, namespace, and the entities in your namespace. For more information, see Managing lifecycles for AWS IoT Things Graph entities, flows, systems, and deployments (p. 144).

Security best practices for AWS IoT Things Graph

AWS IoT Things Graph provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a
complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

To learn about security in AWS IoT, see Security best practices in AWS IoT in the AWS IoT Developer Guide.

For information about hardware security in AWS IoT Greengrass, see Hardware security integration in the AWS IoT Greengrass Developer Guide.

Cloud deployments require you to use an IAM role to execute AWS IoT Things Graph flows on your behalf. AWS IoT Greengrass deployments require you to use an IAM role for AWS IoT Greengrass to execute flows. These roles should have policies that allow AWS IoT Things Graph to perform all of the actions in your flow. Make sure that these roles have only the permissions that they need to execute your flows.

- For information about setting up an environment for cloud deployments, see Prepare for cloud deployments (p. 9).
- For information about setting up an environment for AWS IoT Greengrass deployments, see Setting up your environment for AWS IoT Greengrass deployments (p. 21).
Example device and service definitions

This section contains complete AWS IoT Things Graph device and service definitions written in the AWS IoT Things Graph Data Model.

Each topic in this section shows a single device definition. The definitions include the parent device model, properties, states, actions, events, and capabilities that the device or service requires.

Some of these models are available in the AWS IoT Things Graph console. Each topic notes when that's the case.

Device Definitions (MQTT)
- Aukru HCSR501 motion sensor
- Raspberry Pi camera
- Raspberry Pi camera (for Amazon Rekognition)
- Raspberry Pi screen

Device Definitions (Modbus)
- HW-Group Damocles Mini
- CoolAutomation CoolMasterNet
- Deuta AL-511-00 IP-DALI-BRIDGE V2

Service Definitions
- Rekognition
- getS3Lambda
- wordCount
- saveResponse

Aukru HCSR501 motion sensor

The following GraphQL shows the device definition for the Aukru HC-SR501 motion sensor that is available in the AWS IoT Things Graph console. This device is used in Creating a flow with devices and in Creating a flow with devices and a service.

```graphql
# Motion sensor state.
type MotionSensorState @stateType(id: "urn:tdm:aws/examples:State:MotionSensorState") {
  isMotionDetected: Boolean @property(id: "urn:tdm:aws:property:Boolean")
}

# Property representing the motion sensor state.
type MotionSensorStateProperty @propertyType(id: "urn:tdm:aws/examples:property:MotionSensorStateProperty",
instanceOf: "urn:tdm:aws/examples:State:MotionSensorState",
```
The following GraphQL shows the device definition for the Raspberry Pi camera that is available in the AWS IoT Things Graph console. This device is used in Creating a flow with devices.

```graphql
# Camera state.
type CameraState @stateType(id: "urn:tdm:aws/examples:state:CameraState") {
  lastClickedImage : Uri @property(id: "urn:tdm:aws:property:String")
}

# Property representing the camera state.
type CameraStateProperty @propertyType(id: "urn:tdm:aws/examples:property:CameraStateProperty", instanceOf: "urn:tdm:aws/examples:state:CameraState") {ignore:void}

# The Capture action takes no arguments and returns the state of the device after completing.
type Capture @actionType(id:"urn:tdm:aws/examples:action:Capture") {
  return : CameraStateProperty @property(id: "urn:tdm:aws/examples:property:CameraStateProperty")
}
```
Raspberry Pi camera (for Amazon Rekognition)

The following GraphQL shows the device definition for the Raspberry Pi camera that is available in the AWS IoT Things Graph console. This device is used in Creating a flow with devices and a service.

This camera definition includes a response payload that contains an Amazon Simple Storage Service (Amazon S3) bucket name and the name of an item in the bucket. Amazon Rekognition requires the Amazon S3 bucket and item names as input parameters.

```graphql
# Camera capability.
type CameraCapability @capabilityType(id: "urn:tdm:aws/examples:capability:CameraCapability") {
  STATE: CameraState @state(id: "urn:tdm:aws/examples:state:CameraState")
    capture: Capture @action(id: "urn:tdm:aws/examples:action:Capture")
}

# Camera device model.
type Camera @deviceModel(id: "urn:tdm:aws/examples:deviceModel:Camera", capability: "urn:tdm:aws/examples:capability:CameraCapability"){ignore:void}

# Device definition for the Raspberry Pi Camera.
# In our implementation, this device uses a Java driver to connect to AWS IoT Greengrass.
query RaspberryPiCamera @device(id: "urn:tdm:aws/examples:device:RaspberryPiCamera", deviceModel: "urn:tdm:aws/examples:deviceModel:Camera") {
  MQTT {
    CameraCapability(id: "urn:tdm:aws/examples:capability:CameraCapability") {
      state {
        lastClickedImage(name: "lastImage", property: "urn:tdm:aws:property:String")
      }
      Action(name: "capture") {
        Publish {
          Request(topic: "@macro(${systemRuntime.deviceId}/capture")") {
            params
          }
          Response(topic: "@macro(${systemRuntime.deviceId}/capture/finished") { responsePayload(property: "urn:tdm:aws/examples:property:CameraStateProperty")
          }
        }
      }
    }
  }
}

# Camera state.
type CameraStateRkgnExample @stateType(id: "urn:tdm:aws/examples:state:CameraStateRkgnExample") {
  lastClickedImage : Uri @property(id: "urn:tdm:aws:property:String")
}

# Property representing the camera state.
type CameraStatePropertyRkgnExample @propertyType(id: "urn:tdm:aws/examples:property:CameraStatePropertyRkgnExample", instanceOf: "urn:tdm:aws/examples:state:CameraStateRkgnExample") {ignore:void}
```
# The Capture action takes no arguments and returns the state of the device after completing.
type CaptureRkgnExample @actionType(id:"urn:tdm:aws/examples:action:CaptureRkgnExample") {
    return : CameraStatePropertyRkgnExample @property(id: "urn:tdm:aws/examples:property:CameraStatePropertyRkgnExample")
}

# Camera capability.
type CameraCapabilityRkgnExample @capabilityType(id: "urn:tdm:aws/examples:capability:CameraCapabilityRkgnExample") {
    STATE: CameraStateRkgnExample @state(id: "urn:tdm:aws/examples:state:CameraStateRkgnExample")
    capture: CaptureRkgnExample @action(id: "urn:tdm:aws/examples:action:CaptureRkgnExample")
}

# Camera device model.
type CameraRkgnExample @deviceModel(id: "urn:tdm:aws/examples:deviceModel:CameraRkgnExample", capability: "urn:tdm:aws/examples:capability:CameraCapabilityRkgnExample")

# Device definition for the Raspberry Pi Camera.
# In our implementation, this device uses a Java driver to connect to AWS IoT Greengrass.
query RaspberryPiCameraRkgnExample @device(id: "urn:tdm:aws/examples:device:RaspberryPiCameraRkgnExample", deviceModel: "urn:tdm:aws/examples:deviceModel:CameraRkgnExample") {
  MQTT {
    CameraCapabilityRkgnExample(id: "urn:tdm:aws/examples:capability:CameraCapabilityRkgnExample") {
      state {
        lastClickedImage(name: "lastImage", property: "urn:tdm:aws:property:String"),
        s3BucketName(name: "s3BucketName", property: "urn:tdm:aws:property:String"),
        s3ItemName(name: "s3ItemName", property: "urn:tdm:aws:property:String")
      }
      Action(name: "capture") {
        Publish {
          Request(topic: "macro(${systemRuntime.deviceId}/capture") { params
          }
          Response(topic: "macro(${systemRuntime.deviceId}/capture/finished") {
            responsePayload(property: "urn:tdm:aws/examples:property:CameraStatePropertyRkgnExample")
          }
        }
      }
    }
  }
}

---

**Raspberry Pi screen**

The following GraphQL shows the device definition for the Raspberry Pi screen that is available in the AWS IoT Things Graph console. This device is used in Creating a flow with devices and in Creating a flow with devices and a service.
The following GraphQL shows the device definition for the HW-Group Damocles Mini. This device connects to water, gas, electrical, or other meters, and you can use it in a variety of monitoring scenarios (such as remote equipment monitoring and security and surveillance systems).

For more information about how to create a Modbus device model, see Modbus device modeling 101. That topic explains how to translate details from the product specification into the device definition.
AWS IoT Things Graph User Guide
HW Group Damocles Mini

# Name: Damocles2 MINI
# Manufacturer: HW Group
# Model: Damocles2 MINI

# Action definitions
type GetBoolean @actionType(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean") {
    return: Boolean @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetBoolean @actionType(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetBoolean") {
    inputBoolean: Boolean @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetInputCounter @actionType(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetInputCounter") {
    return: InputCounter @property(id: "urn:tdm:aws:Property:Int32")
}

type SetInputCounter @actionType(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetInputCounter") {
    inputCounterValue: InputCounter @property(id: "urn:tdm:aws:Property:Int32")
}

# Capability definition
type DamoclesMiniCapability @capabilityType(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:capability:DamoclesMiniCapability") {
    getDiscreteInput1Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    getDiscreteInput2Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    getDiscreteInput3Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    getDiscreteInput4Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    getOutput1Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    setOutput1Value: SetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetBoolean")
    getOutput2Value: GetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetBoolean")
    setOutput2Value: SetBoolean @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetBoolean")
    getInput1Counter: GetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetInputCounter")
    setInput1Counter: SetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetInputCounter")
    getInput2Counter: GetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetInputCounter")
    setInput2Counter: SetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetInputCounter")
    getInput3Counter: GetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetInputCounter")
    setInput3Counter: SetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetInputCounter")
    getInput4Counter: GetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:GetInputCounter")
    setInput4Counter: SetInputCounter @action(id: "urn:tdm:aws/examples/modbus/hwgroup/damocles2mini:action:SetInputCounter")
}

# Device Model definition (abstract device)

ignore: void

# Modbus RTU
# Device definition (physical device)
query ModbusDamoclesMini @device(id: "urn:tdm:aws/examples/modbus/hwgroup/ 
hwgroup/damocles2mini:deviceModel:DamoclesMini") {
  Modbus($ServerId: "1") {
    Capability(id: "urn:tdm:aws/examples/modbus/hwgroup/ 
damocles2mini:capability:DamoclesMiniCapability") {
      Action(name: "getDiscreteInput1Value") {
        Params
        ReadDiscreteInputs {
          Request(Address: 100, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
      Action(name: "getDiscreteInput2Value") {
        Params
        ReadDiscreteInputs {
          Request(Address: 101, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
      Action(name: "getDiscreteInput3Value") {
        Params
        ReadDiscreteInputs {
          Request(Address: 102, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
      Action(name: "getDiscreteInput4Value") {
        Params
        ReadDiscreteInputs {
          Request(Address: 103, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
      Action(name: "getOutput1Value") {
        Params
        ReadCoils {
          Request(Address: 200, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
      Action(name: "setOutput1Value") {
        Params {
          param(name: "inputBoolean", property: "urn:tdm:aws:Property:Boolean")
        }
        WriteSingleCoil {
          Request(Address: 200) {
            params {
              param(name: "inputBoolean", property: "urn:tdm:aws:Property:Boolean", value: 
"#{inputBoolean.value}"})
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
          }
        }
      }
    }
  }
}

183
Action(name: "getOutput2Value") {
    Params
    ReadCoils {
        Request(Address: 201, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "setOutput2Value") {
    Params {
        param(name: "inputBoolean", property: "urn:tdm:aws:Property:Boolean")
    }
    WriteSingleCoil {
        Request(Address: 201) {
            params {
                param(name: "inputBoolean", property: "urn:tdm:aws:Property:Boolean", value: 
"#${inputBoolean.value}")
            }
        }
    }
}

Action(name: "getInput1Counter") {
    Params
    ReadInputRegisters {
        Request(Address: 201, ReadCount: 2)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int32")
        }
    }
}

Action(name: "setInput1Counter") {
    Params {
        param(name: "inputCounterValue", property: "urn:tdm:aws:Property:Int32")
    }
    WriteMultipleRegisters {
        Request(Address: 201, WriteCount: 2) {
            params {
                param(name: "inputCounterValue", property: "urn:tdm:aws:Property:Int32", value: 
"#${inputCounterValue.value}")
            }
        }
    }
}

Action(name: "getInput2Counter") {
    Params
    ReadInputRegisters {
        Request(Address: 203, ReadCount: 2)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int32")
        }
    }
}

Action(name: "setInput2Counter") {
    Params {
        param(name: "inputCounterValue", property: "urn:tdm:aws:Property:Int32")
    }
    WriteMultipleRegisters {
        Request(Address: 203, WriteCount: 2) {
            params {
                param(name: "inputCounterValue", property: "urn:tdm:aws:Property:Int32", value: 
"#${inputCounterValue.value}")
            }
        }
    }
}
The following GraphQL shows the device definition for the CoolAutomation CoolMasterNet. This device is a communication bridge that connects home automation systems with heating, ventilation, and air conditioning systems. Therefore, this device definition is just a template. You can rename the actions based on the resources of the device that's connected to the bridge.

For more information about how to create a Modbus device model, see Modbus device modeling 101. That topic explains how to translate details from the product specification into the device definition.
The CoolMasterNet uses virtual addresses (VAs). In this definition, the register addresses are examples. You must update them to reflect your own VA configuration. For more information, see the VA's Configuration section in the product specification.

```xml
{  
    # Manufacturer: Cool Automation  
    # Model: CoolMasterNet  

    # Action definitions

type GetOperationMode @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetOperationMode"){
    return: OperationMode @property(id: "urn:tdm:aws:Property:Int16")
}

type SetOperationMode @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetOperationMode"){
    inputOperationMode: OperationMode @property(id: "urn:tdm:aws:Property:Int16")
}

type GetFanSpeed @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetFanSpeed"){
    return: FanSpeed @property(id: "urn:tdm:aws:Property:Int16")
}

type SetFanSpeed @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetFanSpeed"){
    inputFanSpeed: FanSpeed @property(id: "urn:tdm:aws:Property:Int16")
}

type SetTemperature @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetTemperature"){
    inputTemperature: Temperature @property(id: "urn:tdm:aws:Property:Int16")
}

type GetTemperatureSetting @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetTemperatureSetting"){
    return: Temperature @property(id: "urn:tdm:aws:Property:Int16")
}

type GetRoomTemperature @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetRoomTemperature"){
    return: Temperature @property(id: "urn:tdm:aws:Property:Int16")
}

type GetSwitchState_H @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetSwitchState_H"){
    return: SwitchOnOff_H @property(id: "urn:tdm:aws:Property:Int16")
}

type SetSwitchState_H @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetSwitchState_H"){
    inputSwitchOnOff_H: SwitchOnOff_H @property(id: "urn:tdm:aws:Property:Int16")
}

type GetSwitchState_C @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetSwitchState_C"){
    return: SwitchOnOff_C @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetSwitchState_C @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetSwitchState_C"){
    inputSwitchOnOff_C: SwitchOnOff_C @property(id: "urn:tdm:aws:Property:Boolean")
}
```
type GetSwingSetting @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetSwingSetting") {
  return: SwingSetting @property(id: "urn:tdm:aws:Property:Int16")
}

type SetSwingSetting @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetSwingSetting") {
  inputSwingSetting: SwingSetting @property(id: "urn:tdm:aws:Property:Int16")
}

type GetFilterSign_H @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetFilterSign_H") {
  return: FilterSign_H @property(id: "urn:tdm:aws:Property:Int16")
}

type SetFilterSign_H @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetFilterSign_H") {
  inputFilterSign_H: FilterSign_H @property(id: "urn:tdm:aws:Property:Int16")
}

type GetFilterSign_C @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetFilterSign_C") {
  return: FilterSign_C @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetFilterSign_C @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetFilterSign_C") {
  inputFilterSign_C: FilterSign_C @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetExternalTerminalsStatus @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetExternalTerminalsStatus") {
  return: ExternalTerminalsStatus @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetInhibitIndoorUnitOnOperationSetting @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetInhibitIndoorUnitOnOperationSetting") {
  return: InhibitIndoorUnitOnOperationSetting @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetInhibitIndoorUnitOnOperationSetting @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetInhibitIndoorUnitOnOperationSetting") {
  inputInhibitIndoorUnitOnOperationSetting: InhibitIndoorUnitOnOperationSetting @property(id: "urn:tdm:aws:Property:Int16")
}

type GetAnalogInput1 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetAnalogInput1") {
  return: AnalogInput1 @property(id: "urn:tdm:aws:Property:Int16")
}

type GetAnalogInput2 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetAnalogInput2") {
  return: AnalogInput2 @property(id: "urn:tdm:aws:Property:Int16")
}

type GetDigitalOutput1 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput1") {
  return: DigitalOutput1 @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetDigitalOutput2 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput2") {
  return: DigitalOutput2 @property(id: "urn:tdm:aws:Property:Boolean")
}
type GetDigitalOutput3 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput3") {
  return: DigitalOutput3 @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetDigitalOutput4 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput4") {
  return: DigitalOutput4 @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetDigitalOutput5 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput5") {
  return: DigitalOutput5 @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetDigitalOutput6 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalOutput6") {
  return: DigitalOutput6 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput1 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput1") {
  inputDigitalOutput1: DigitalOutput1 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput2 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput2") {
  inputDigitalOutput2: DigitalOutput2 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput3 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput3") {
  inputDigitalOutput3: DigitalOutput3 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput4 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput4") {
  inputDigitalOutput4: DigitalOutput4 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput5 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput5") {
  inputDigitalOutput5: DigitalOutput5 @property(id: "urn:tdm:aws:Property:Boolean")
}

type SetDigitalOutput6 @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetDigitalOutput6") {
  inputDigitalOutput6: DigitalOutput6 @property(id: "urn:tdm:aws:Property:Boolean")
}

type GetDigitalInput @actionType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput") {
  return: DigitalInput @property(id: "urn:tdm:aws:Property:Boolean")
}

# Capability definition
type CoolMasterNetCapability @capabilityType(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:capability:CoolMasterNetCapability") {
  getOperationMode: GetOperationMode @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetOperationMode")
  setOperationMode: SetOperationMode @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:SetOperationMode")
  getFanSpeed: GetFanSpeed @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetFanSpeed")
setFanSpeed: SetFanSpeed @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetFanSpeed")
setTemperature: SetTemperature @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetTemperature")
getTemperatureSetting: GetTemperatureSetting @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:GetTemperatureSetting")
getRoomTemperature_H: GetRoomTemperature @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:GetRoomTemperature")
getRoomTemperature_I: GetRoomTemperature @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:GetRoomTemperature")
getSwitchState_H: GetSwitchState_H @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:GetSwitchState_H")
setSwitchState_H: SetSwitchState_H @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:SetSwitchState_H")
setSwitchState_C: SetSwitchState_C @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:SetSwitchState_C")
setSwitchState_C: SetSwitchState_C @action(id: "urn:tdm:aws/examples/modbus/
coolautomation/coolmasternet:action:SetSwitchState_C")
setSwingSetting: SetSwingSetting @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetSwingSetting")
getFilterSign_H: GetFilterSign_H @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetFilterSign_H")
setFilterSign_H: SetFilterSign_H @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetFilterSign_H")
setFilterSign_H: SetFilterSign_H @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetFilterSign_H")
getExternalTerminalsStatus: GetExternalTerminalsStatus @action(id: "urn:tdm:aws/examples/
modbus/coolautomation/coolmasternet:action:GetExternalTerminalsStatus")
getInhibitIndoorUnitOnOperationSetting: GetInhibitIndoorUnitOnOperationSetting
@action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetInhibitIndoorUnitOnOperationSetting")
getInhibitIndoorUnitOnOperationSetting: GetInhibitIndoorUnitOnOperationSetting
@action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetInhibitIndoorUnitOnOperationSetting")
getAnalogInput1: GetAnalogInput1 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetAnalogInput1")
getDigitalOutput1: GetDigitalOutput1 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput1")
getDigitalOutput2: GetDigitalOutput2 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput2")
getDigitalOutput3: GetDigitalOutput3 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput3")
getDigitalOutput4: GetDigitalOutput4 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput4")
getDigitalOutput5: GetDigitalOutput5 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput5")
getDigitalOutput6: GetDigitalOutput6 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:GetDigitalOutput6")
setDigitalOutput1: SetDigitalOutput1 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput1")
setDigitalOutput2: SetDigitalOutput2 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput2")
setDigitalOutput3: SetDigitalOutput3 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput3")
setDigitalOutput4: SetDigitalOutput4 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput4")
setDigitalOutput5: SetDigitalOutput5 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput5")
setDigitalOutput6: SetDigitalOutput6 @action(id: "urn:tdm:aws/examples/modbus/coolautomation/
coolmasternet:action:SetDigitalOutput6")
getDigitalInput1: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")
getDigitalInput2: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")
getDigitalInput3: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")
getDigitalInput4: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")
getDigitalInput5: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")
getDigitalInput6: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:action:GetDigitalInput")

# Device Model definition (abstract device)
type CoolMasterNet @deviceModel(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:deviceModel:CoolMasterNet", capability: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:capability:CoolMasterNetCapability") {
  ignore: void
}

# Modbus RTU
# Device definition (physical device)
  Modbus(ServerId: "1") {
    Capability(id: "urn:tdm:aws/examples/modbus/coolautomation/coolmasternet:capability:CoolMasterNetCapability") {
      Action(name: "getOperationMode") {
        Params
        ReadHoldingRegisters {
          Request(Address: 0, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
          }
        }
      }
    }
    Action(name: "setOperationMode") {
      Params {
        param(name: "inputOperationMode", property: "urn:tdm:aws:Property:Int16")
      }
      WriteSingleRegister {
        Request(Address: 0) {
          params {
            param(name: "inputOperationMode", property: "urn:tdm:aws:Property:Int16", value: "${inputOperationMode.value}")
          }
        }
      }
    }
    Action(name: "getFanSpeed") {
      Params
      ReadHoldingRegisters {
        Request(Address: 1, ReadCount: 1)
        Response {
          responsePayload(property: "urn:tdm:aws:Property:Int16")
        }
      }
    }
    Action(name: "setFanSpeed") {
      Params {
        param(name: "inputFanSpeed", property: "urn:tdm:aws:Property:Int16")
      }
      WriteSingleRegister {
        Request(Address: 1) {
        }
      }
    }
  }
}

190
params {
    param(name: "inputFanSpeed", property: "urn:tdm:aws:Property:Int16", value: 
        "${inputFanSpeed.value}""
    )
} }

Action(name: "setTemperature") {
    Params {
        param(name: "inputTemperature", property: "urn:tdm:aws:Property:Int16")
    }
    WriteSingleRegister {
        Request(Address: 2) {
            params {
                param(name: "inputTemperature", property: "urn:tdm:aws:Property:Int16", 
                    value: 
                        "${inputTemperature.value}""
                )
            }
        }
    }
}

Action(name: "getTemperatureSetting") {
    Params
    ReadHoldingRegisters {
        Request(Address: 2, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
        }
    }
}

Action(name: "getRoomTemperature_H") {
    Params
    ReadHoldingRegisters {
        Request(Address: 6, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
        }
    }
}

Action(name: "getRoomTemperature_I") {
    Params
    ReadInputRegisters {
        Request(Address: 1, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
        }
    }
}

Action(name: "getSwitchState_H") {
    Params
    ReadHoldingRegisters {
        Request(Address: 3, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
        }
    }
}

Action(name: "setSwitchState_H") {
    Params {
        param(name: "inputSwitchOnOff_H", property: "urn:tdm:aws:Property:Int16")
    }
    WriteSingleRegister {
        Request(Address: 3) {
            params {
                param(name: "inputSwitchOnOff_H", property: "urn:tdm:aws:Property:Int16", 
                    value: 
                        "${inputSwitchOnOff_H.value}""
                )
            }
        }
    }
}
Action(name: "getSwitchState_C") {
  Params
  ReadCoils {
    Request(Address: 0, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "setSwitchState_C") {
  Params {
    WriteSingleCoil {
      Request(Address: 0) {
        params {
          param(name: "inputSwitchOnOff_C", property: "urn:tdm:aws:Property:Boolean", value: 
                          "#{inputSwitchOnOff_C.value}")
        }
      }
    }
  }
}

Action(name: "getSwingSetting") {
  Params
  ReadHoldingRegisters {
    Request(Address: 5, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setSwingSetting") {
  Params {
    WriteSingleRegister {
      Request(Address: 5) {
        params {
          param(name: "inputSwingSetting", property: "urn:tdm:aws:Property:Int16", value: 
                          "#{inputSwingSetting.value}")
        }
      }
    }
  }
}

Action(name: "getFilterSign_H") {
  Params
  ReadHoldingRegisters {
    Request(Address: 4, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setFilterSign_H") {
  Params {
    WriteSingleRegister {
      Request(Address: 4) {
        params {
          param(name: "inputFilterSign_H", property: "urn:tdm:aws:Property:Int16", value: 
                          "#{inputFilterSign_H.value}")
        }
      }
    }
  }
}
Action(name: "getFilterSign_C") {
  Params
  ReadCoils {
    Request(Address: 1, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "setFilterSign_C") {
  Params {
    param(name: "inputFilterSign_C", property: "urn:tdm:aws:Property:Boolean")
  }
  WriteSingleCoil {
    Request(Address: 1) {
      params {
        param(name: "inputFilterSign_C", property: "urn:tdm:aws:Property:Boolean",
          value: "${inputFilterSign_C.value}")
      }
    }
  }
}

Action(name: "getExternalTerminalsStatus") {
  Params
  ReadCoils {
    Request(Address: 2, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getInhibitIndoorUnitOnOperationSetting") {
  Params
  ReadCoils {
    Request(Address: 3, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "setInhibitIndoorUnitOnOperationSetting") {
  Params {
    param(name: "inputInhibitIndoorUnitOnOperationSetting", property: "urn:tdm:aws:Property:Int16")
  }
  WriteSingleCoil {
    Request(Address: 3) {
      params {
        param(name: "inputInhibitIndoorUnitOnOperationSetting", property: "urn:tdm:aws:Property:Int16",
          value: "${inputInhibitIndoorUnitOnOperationSetting.value}"")
      }
    }
  }
}

Action(name: "getAnalogInput1") {
  Params
  ReadInputRegisters {
    Request(Address: 13, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}
Action(name: "getAnalogInput2") {
  Params
  ReadInputRegisters {
    Request(Address: 14, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getDigitalOutput1") {
  Params
  ReadCoils {
    Request(Address: 9, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getDigitalOutput2") {
  Params
  ReadCoils {
    Request(Address: 10, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getDigitalOutput3") {
  Params
  ReadCoils {
    Request(Address: 11, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getDigitalOutput4") {
  Params
  ReadCoils {
    Request(Address: 12, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getDigitalOutput5") {
  Params
  ReadCoils {
    Request(Address: 13, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "getDigitalOutput6") {
  Params
  ReadCoils {
    Request(Address: 14, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Boolean")
    }
  }
}

Action(name: "setDigitalOutput1") {
  Params {
    param(name: "inputDigitalOutput1", property: "urn:tdm:aws:Property:Boolean")
  }
}
Action(name: "setDigitalOutput1") {
  Params {
    param(name: "inputDigitalOutput1", property: "urn:tdm:aws:Property:Boolean",
    value: "${inputDigitalOutput1.value}"
  }
  WriteSingleCoil {
    Request(Address: 9) {
      params {
        param(name: "inputDigitalOutput1", property: "urn:tdm:aws:Property:Boolean",
        value: "${inputDigitalOutput1.value}"))
      }
    }
  }
}

Action(name: "setDigitalOutput2") {
  Params {
    param(name: "inputDigitalOutput2", property: "urn:tdm:aws:Property:Boolean")
  }
  WriteSingleCoil {
    Request(Address: 10) {
      params {
        value: "${inputDigitalOutput2.value}"
      }
    }
  }
}

Action(name: "setDigitalOutput3") {
  Params {
    param(name: "inputDigitalOutput3", property: "urn:tdm:aws:Property:Boolean")
  }
  WriteSingleCoil {
    Request(Address: 11) {
      params {
        value: "${inputDigitalOutput3.value}"
      }
    }
  }
}

Action(name: "setDigitalOutput4") {
  Params {
    param(name: "inputDigitalOutput4", property: "urn:tdm:aws:Property:Boolean")
  }
  WriteSingleCoil {
    Request(Address: 12) {
      params {
        value: "${inputDigitalOutput4.value}"
      }
    }
  }
}

Action(name: "setDigitalOutput5") {
  Params {
    param(name: "inputDigitalOutput5", property: "urn:tdm:aws:Property:Boolean")
  }
  WriteSingleCoil {
    Request(Address: 13) {
      params {
        value: "${inputDigitalOutput5.value}"
      }
    }
  }
}

Action(name: "setDigitalOutput6") {
  Params {
    param(name: "inputDigitalOutput6", property: "urn:tdm:aws:Property:Boolean")
  }
WriteSingleCoil {
    Request(Address: 14) {
        params {
            value: "#{inputDigitalOutput6.value}"
        )
    }
}
}

Action(name: "getDigitalInput1") {
    Params
    ReadDiscreteInputs {
        Request(Address: 9, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "getDigitalInput2") {
    Params
    ReadDiscreteInputs {
        Request(Address: 10, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "getDigitalInput3") {
    Params
    ReadDiscreteInputs {
        Request(Address: 11, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "getDigitalInput4") {
    Params
    ReadDiscreteInputs {
        Request(Address: 12, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "getDigitalInput5") {
    Params
    ReadDiscreteInputs {
        Request(Address: 13, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}

Action(name: "getDigitalInput6") {
    Params
    ReadDiscreteInputs {
        Request(Address: 14, ReadCount: 1)
        Response {
            responsePayload(property: "urn:tdm:aws:Property:Boolean")
        }
    }
}
Deuta AL-511-00 IP-DALI-BRIDGE V2

The following GraphQL shows the device definition for the Deuta AL-511-00 IP-DALI-BRIDGE V2. This device is a communication bridge that connects a controller with an Ethernet interface to any DALI device. Therefore, this device definition is just a template. You can rename the actions based on the resources of the device that's connected to the bridge.

For more information about how to create a Modbus device model, see Modbus device modeling 101. That topic explains how to translate details from the product specification into the device definition.

```graphql
{
  # Manufacturer: Deuta
  # Model: AL-511-00 IP-DALI-BRIDGE V2
  # Action definitions
  type GetVersion @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetVersion") {
    return: VersionNumber @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetSerialNumber @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetSerialNumber") {
    return: SerialNumber @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetAnalogInputVoltage @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputVoltage") {
    return: AnalogInputVoltage @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetAnalogInputPercent @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputPercent") {
    return: Percent @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetDigitalThresholdValue @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalThresholdValue") {
    return: DigitalThresholdValue @property(id: "urn:tdm:aws:Property:Int16")
  }
  type SetDigitalThresholdValue @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalThresholdValue") {
    inputDigitalThresholdValue: DigitalThresholdValue @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetDigitalInput @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalInput") {
    return: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
  }
  type GetDigitalOutputTemp @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalOutputTemp") {
    return: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
  }
  type SetDigitalOutputTemp @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalOutputTemp") {
  

```
inputDigitalOutputTemp: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
}
type GetDigitalOutputRemanent @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalOutputRemanent") {
  return: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
}
type SetDigitalOutputRemanent @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalOutputRemanent") {
  inputDigitalOutputRemanent: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
}
type GetChipTemperature @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetChipTemperature") {
  return: ChipTemperature @property(id: "urn:tdm:aws:Property:Int16")
}
type GetAnalogInputOffset @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputOffset") {
  return: AnalogInputOffset @property(id: "urn:tdm:aws:Property:Int16")
}
type SetAnalogInputOffset @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetAnalogInputOffset") {
  inputAnalogInputOffset: AnalogInputOffset @property(id: "urn:tdm:aws:Property:Int16")
}
type GetAnalogInputGradient @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputGradient") {
  return: AnalogInputGradient @property(id: "urn:tdm:aws:Property:Int16")
}
type SetAnalogInputGradient @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetAnalogInputGradient") {
  inputAnalogInputGradient: AnalogInputGradient @property(id: "urn:tdm:aws:Property:Int16")
}
type GetSupplyVoltageStatus @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetSupplyVoltageStatus") {
  return: DigitalSignal @property(id: "urn:tdm:aws:Property:Int16")
}
type SetDALIcommand @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDALIcommand") {
  inputCommand: DALIcommand @property(id: "urn:tdm:aws:Property:Int16")
}
type GetDALIanswer @actionType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDALIanswer") {
  return: DALIanswer @property(id: "urn:tdm:aws:Property:Int16")
}

# Capability definition
type DALILightControllerCapability @capabilityType(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:capability:DALILightControllerCapability") {
  getMainVersion_HW: GetVersion @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetVersion")
  getSubVersion_HW: GetVersion @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetVersion")
  getMainVersion_SW: GetVersion @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetVersion")
  getSubVersion_SW: GetVersion @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetVersion")
  getSerialNumber_LSB: GetSerialNumber @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetSerialNumber")
getSerialNumber_MSB: GetSerialNumber @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetSerialNumber")
getAnalogInputVoltage: GetAnalogInputVoltage @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputVoltage")
getAnalogInputPercent: GetAnalogInputPercent @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputPercent")
getDigitalThresholdValue_low: GetDigitalThresholdValue @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalThresholdValue")
setDigitalThresholdValue_low: SetDigitalThresholdValue @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalThresholdValue")
getDigitalThresholdValue_high: GetDigitalThresholdValue @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalThresholdValue")
setDigitalThresholdValue_high: SetDigitalThresholdValue @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalThresholdValue")
getDigitalInput: GetDigitalInput @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalInput")
getDigitalOutputTemp: GetDigitalOutputTemp @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalOutputTemp")
setDigitalOutputTemp: SetDigitalOutputTemp @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalOutputTemp")
getDigitalOutputRemanent: GetDigitalOutputRemanent @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDigitalOutputRemanent")
setDigitalOutputRemanent: SetDigitalOutputRemanent @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDigitalOutputRemanent")
getChipTemperature: GetChipTemperature @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetChipTemperature")
getAnalogInputOffset: GetAnalogInputOffset @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputOffset")
setAnalogInputOffset: SetAnalogInputOffset @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetAnalogInputOffset")
getAnalogInputGradient: GetAnalogInputGradient @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetAnalogInputGradient")
setAnalogInputGradient: SetAnalogInputGradient @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetAnalogInputGradient")
getSupplyVoltageStatus: GetSupplyVoltageStatus @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetSupplyVoltageStatus")
setDALIcommand: SetDALIcommand @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:SetDALIcommand")
getDALIanswer: GetDALIanswer @action(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:action:GetDALIanswer")

# Device Model definition (abstract device)
type DALILightController @deviceModel(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:deviceModel:DALILightController", capability: "urn:tdm:aws/examples/modbus/deuta/dalibridge:capability:DALILightControllerCapability") {
  ignore: void
}

# Device definition (physical device) query ModbusDALILightController @device(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:device:ModbusDALILightController", deviceModel: "urn:tdm:aws/examples/modbus/deuta/dalibridge:deviceModel:DALILightController") {
  Modbus(ServerId: "1") {
    Capability(id: "urn:tdm:aws/examples/modbus/deuta/dalibridge:capability:DALILightControllerCapability") {
      Action(name: "getMainVersion_HW") {
        Params
        ReadHoldingRegisters {
          Request(Address: 2, ReadCount: 1)
          Response {
            responsePayload(property: "urn:tdm:aws:Property:Int16")
          }
        }
      }
    }
  }
}
Action(name: "getSubVersion_HW") {
  Params
  ReadHoldingRegisters {
    Request(Address: 3, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getMainVersion_SW") {
  Params
  ReadHoldingRegisters {
    Request(Address: 4, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getSubVersion_SW") {
  Params
  ReadHoldingRegisters {
    Request(Address: 5, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getSerialNumber_LSB") {
  Params
  ReadHoldingRegisters {
    Request(Address: 6, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getSerialNumber_MSB") {
  Params
  ReadHoldingRegisters {
    Request(Address: 7, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getAnalogInputVoltage") {
  Params
  ReadHoldingRegisters {
    Request(Address: 24, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getAnalogInputPercent") {
  Params
  ReadHoldingRegisters {
    Request(Address: 25, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getDigitalThresholdValue_low") {
  Params
  ReadHoldingRegisters {

Action(name: "setDigitalThresholdValue_low") {
  Params {
    param(name: "inputDigitalThresholdValue", property: "urn:tdm:aws:Property:Int16")
  }
  WriteSingleRegister {
    Request(Address: 26) {
      params {
      }
    }
  }
}

Action(name: "getDigitalThresholdValue_high") {
  Params
  ReadHoldingRegisters {
    Request(Address: 27, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setDigitalThresholdValue_high") {
  Params {
    param(name: "inputDigitalThresholdValue", property: "urn:tdm:aws:Property:Int16")
  }
  WriteSingleRegister {
    Request(Address: 27) {
      params {
      }
    }
  }
}

Action(name: "getDigitalInput") {
  Params
  ReadHoldingRegisters {
    Request(Address: 28, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getDigitalOutputTemp") {
  Params
  ReadHoldingRegisters {
    Request(Address: 29, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setDigitalOutputTemp") {
  Params {
    param(name: "inputDigitalOutputTemp", property: "urn:tdm:aws:Property:Int16")
  }
  WriteSingleRegister {
    Request(Address: 29) {
      params {

})

Action(name: "getDigitalOutputRemanent") {
  Params
  ReadHoldingRegisters {
    Request(Address: 30, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setDigitalOutputRemanent") {
  Params {
  }
  WriteSingleRegister {
    Request(Address: 30) {
      params {
      }
    }
  }
}

Action(name: "getChipTemperature") {
  Params
  ReadHoldingRegisters {
    Request(Address: 31, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "getAnalogInputOffset") {
  Params
  ReadHoldingRegisters {
    Request(Address: 32, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}

Action(name: "setAnalogInputOffset") {
  Params {
    param(name: "inputAnalogInputOffset", property: "urn:tdm:aws:Property:Int16")
  }
  WriteSingleRegister {
    Request(Address: 32) {
      params {
      }
    }
  }
}

Action(name: "getAnalogInputGradient") {
  Params
  ReadHoldingRegisters {
    Request(Address: 33, ReadCount: 1)
    Response {
      responsePayload(property: "urn:tdm:aws:Property:Int16")
    }
  }
}
Amazon Rekognition DetectFaces API

The following GraphQL shows the service definition for the Amazon Rekognition DetectFaces API that is available in the AWS IoT Things Graph console. This service is used in Creating a flow with devices and a service.

```graphql
# Amazon Rekognition DetectFaces action. This action takes Amazon S3 bucket and item names as input and
```
# AWS IoT Things Graph User Guide

## getS3Lambda

The following GraphQL shows the definition for the **getS3Lambda** service that is available in the AWS IoT Things Graph console. This service is used in Creating a flow with Lambda functions.

```graphql
# The getS3ObjectDocument state defines the fields in the response of the getS3ObjectAsString action.
# It contains the contents of a file as a string and the length of the string.
type S3ObjectDocument @stateType(id: "urn:tdm:aws/examples:State:S3ObjectDocument") {
    message: String @property(id: "urn:tdm:aws:property:String"),
    length: Int @property(id: "urn:tdm:aws:property:Int32")
}
```
The following GraphQL shows the definition for the `wordCount` service that is available in the AWS IoT Things Graph console. This service is used in Creating a flow with Lambda functions.
# The wordCountResponseStructure state defines the structure of the wordCountAction response.
type wordCountResponseStructure @stateType(id: "urn:tdm:aws/examples:State:wordCountResponseStructure") {
  uniqueWordsCount: Int @property(id: "urn:tdm:aws:Property:Int32"),
  mostFrequentWord: String @property(id: "urn:tdm:aws:property:String"),
  mostFrequentWordFrequency: Int @property(id: "urn:tdm:aws:Property:Int32")
}

# Property representing the wordCountResponseStructure state.
type wordCountResponse @propertyType(id: "urn:tdm:aws/examples:property:wordCountResponse"
instanceOf: "urn:tdm:aws/examples:State:wordCountResponseStructure") {ignore:void}

# The wordCountAction takes a string as input and returns the number of unique words and the
# most frequently used word in the string.
type wordCountAction @actionType(id: "urn:tdm:aws/examples:action:wordCountAction") {
  message: String @property(id: "urn:tdm:aws:property:String"),
  return: wordCountResponse @property(id: "urn:tdm:aws/examples:property:wordCountResponse")
}

# The wordCount capability.
type wordCountCapability @capabilityType(id: "urn:tdm:aws/examples:capability:wordCountCapability") {
  wordCount : wordCountAction @action(id: "urn:tdm:aws/examples:action:wordCountAction")
}

# Service definition for wordCount. It's implemented as an AWS IoT Greengrass Lambda function.
query wordCount @service(id: "urn:tdm:aws/examples:Service:wordCountLambda") {
  AwsLambda {
    wordCountCapability(id: "urn:tdm:aws/examples:capability:wordCountCapability") {
      Action(name: "wordCount") {
        params {
          param(name: "message", property:"urn:tdm:aws:property:String")
        }
        InvokeGreengrassLambda {
          Request(arn: "$macro(arn:aws:lambda:${systemConfig.awsRegion}:${systemConfig.awsAccountId}:function:WordCount:1") {
            params {
              param(name: "message", property:"urn:tdm:aws:property:String", value: "${message.value}")
            }
            Response {
              responsePayload(property: "urn:tdm:aws/examples:property:wordCountResponse")
            }
          }
        }
      }
    }
  }
}

saveResponse

The following GraphQL shows the definition for the saveResponse service that is available in the AWS IoT Things Graph console. This service is used in Creating a flow with Lambda functions.
This definition assumes that the `wordCount` service is already defined.

```graphql
# The saveResponseAction takes the response of the wordCount service as input and saves it to a location
# in Amazon S3 that is specified in the AWS Lambda function that the service exposes.
type saveResponseAction @actionType(id: "urn:tdm:aws/examples:action:saveResponseAction") {
  response: wordCountResponse @property(id: "urn:tdm:aws/examples:property:wordCountResponse")
}

# The saveResponse capability.
type saveResponseCapability @capabilityType(id: "urn:tdm:aws/examples:capability:saveResponseCapability") {
  save: saveResponseAction @action(id: "urn:tdm:aws/examples:action:saveResponseAction")
}

# Service definition for saveResponse. It’s implemented as an AWS IoT Greengrass Lambda function.
query saveResponse @service(id: "urn:tdm:aws/examples:Service:saveResponseLambda") {
  AwsLambda {
    saveResponseCapability(id: "urn:tdm:aws/examples:capability:saveResponseCapability") {
      Action(name: "save") {
        params {
          param(name: "response", property: "urn:tdm:aws/examples:property:wordCountResponse")
        }
      }
    }
    InvokeGreengrassLambda {
      Request(arn: "$macro(arn:aws:lambda:${systemConfig.awsRegion}:${systemConfig.awsAccountId}:function:SaveToS3:1)") {
        params {
          param(name: "response", property: "urn:tdm:aws/examples:property:wordCountResponse", value: "${response.value}")
        }
      }
    }
  }
}
```
Deploying and running AWS IoT Things Graph on AWS IoT Greengrass

This topic explains how AWS IoT Things Graph interacts with your AWS IoT Greengrass group when you deploy your flow configurations to your AWS IoT Greengrass core. It also describes how you can monitor and debug your flows after they start running in your AWS IoT Greengrass group.

AWS IoT Greengrass deployments

AWS IoT Things Graph is deployed as an AWS IoT Greengrass connector on an AWS IoT Greengrass core device. You deploy a flow configuration by using the DeploySystemInstance API. You must specify an Amazon Simple Storage Service (Amazon S3) bucket and an AWS IoT Greengrass group.

AWS IoT Things Graph then performs the following actions on these two resources:

- Generates a deployment artifact containing the dependency closure of the flow configuration, and saves it to the Amazon S3 bucket. The file name of this artifact is a concatenation of `flow-name.tar.gz`. If you delete this file after the deployment is complete, your existing deployment will work, but redeployments won't.
- Deploys the AWS IoT Things Graph connector to the AWS IoT Greengrass group and configures it to use the configuration stored in the S3 bucket.
- Adds the devices specified in the flow configuration.
- Subscribes the AWS IoT Things Graph connector and the devices to the MQTT topics specified in the device definitions.

**Note**
You must create a `thingsgraph` directory at the root of your AWS IoT Greengrass core device. Deployments to your core device won't work if this directory doesn't exist. Don't modify the files that are put in this directory. For more information on setting up your AWS IoT Greengrass core, see Setting up your environment for AWS IoT Greengrass deployments.

After your first deployment of a flow configuration to your AWS IoT Greengrass core device, AWS IoT Things Graph deploys the AWS IoT Things Graph connector to your AWS IoT Greengrass core device. AWS IoT Things Graph also installs a database file named `ThingsGraph.db` and a directory named `engine` to the `thingsgraph` directory that you created when you set up your AWS IoT Greengrass core.

The `ThingsGraph.log` file appears in the `/greengrass/ggc/var/log/user/Region/ThingsGraph` directory. This log file is your primary resource for debugging your AWS IoT Things Graph flows as they run in your group.

For more information about the steps that are required to deploy AWS IoT Things Graph and flows to an AWS IoT Greengrass group, see Creating and deploying flows. For more information about setting up your AWS IoT Greengrass group for AWS IoT Things Graph, see Setting up your environment.
Verifying flows in your AWS IoT Greengrass group

After you deploy a flow for the first time, AWS IoT Things Graph installs the AWS IoT Things Graph connector in your AWS IoT Greengrass core and creates a log file named ThingsGraph.log. If the connector installs correctly and starts running, you'll see the following log entries in the ThingsGraph.log file.

```plaintext
[INFO]-bootstrapping DB
[INFO]-bootstrapping entities
[INFO]-bootstrapping entities done
[INFO]-bootstrapping DB done
[INFO]-FlowsBootstrap: registered SDC: urn:tdm:Region/Account ID/
default:Deployment:MyExample01_V001
[INFO]-Initialized ThingsGraph Runtime vx.x.x
```

The log entries that begin with bootstrapping indicate that the ThingsGraph.db file is being created and populated by entities, as specified by the dependency closure that AWS IoT Things Graph has uploaded to your Amazon S3 bucket. These entities include the device models, systems, and flow templates that you specified in your system flow configuration (SDC).

The FlowsBootstrap entry indicates that your SDC has been added to your AWS IoT Greengrass group. The Initialized ThingsGraph Runtime entry indicates that AWS IoT Things Graph has successfully installed and is running on your core device.

Later entries in the log file relate to the execution of the flow. These two entries indicate that the trigger condition has been met and the flow execution has begun.

```plaintext
[INFO]-Trigger expression condition: [every 10 seconds] met?  true
[INFO]-FlowsSchedulerClientGreengrassImpl: StartFlowExecution:
  flowInstanceId:urn:tdm:Region/Account ID/
  flowType:urn:tdm:Region/Account ID/default:Workflow:MyExample01_V001
  flowParams:{"device1Id":"DeutaBridgeV2"}
```

The bracketed text in the first entry contains the trigger condition logic that you specified for your flow (a time interval in this case). The FlowsSchedulerClientGreengrassImpl: StartFlowExecution: entry indicates that the flow has begun executing. The following three entries contain information about this instance of the flow. The value of flowInstanceId is a concatenation of the flow ID (a URN), the timestamp, the name of the trigger, and a universally unique identifier (UUID). The value of flowParams contains the parameters that have been passed to the flow (the device ID in this case).

After the flow execution begins, a series of StateTransition entries appears in the log. The first occurs at the very beginning, and indicates that the StartFlowExecutionTask has been performed.

```plaintext
[INFO]-[StateTransition-urn:tdm:Region/Account ID/
default:Workflow:MyExample03_V001_2019-03-12T2
0:55:37.041Z_TenSecondTrigger_2ba7d218-2b9e-4005-ad96-8779d589dac2]
  ExecutionStarted(super=StateTransitionMessage(super=Message
(super=com.amazonaws.iotflowengine.messages.ExecutionStarted@93227493, messageId=1939, 
flowInstanceId=urn:tdm:Region/Account ID/
default:Workflow:MyExample03_V001_2019-03-12T2:55:37.041Z_TenSecondTrigger_2ba7d218-2b9e-4005-ad96-8779d589dac2, 
```

209
StateTransition entries include the same values for flowInstanceId and flowParams that occur in the preceding entries.

After each step in the flow is executed, corresponding log entries for StateTransition appear in the log file, as in the following example for step 2 of a flow.

```
StepSucceeded(super=StateTransitionMessage(super=Message(super=com.amazonaws.iotflowengine.messages.StepSucceeded@54bcaae5,
messageId=12053, flowInstanceId=urn:tdm:Region/Account ID/default:Workflow:MyExample03_V001_2019-03-12T21:12:08.661Z_TenSecondTrigger_b190369e-6b44-4d14-8468-7d5c1876893a,
createdAt=1552425129450)),
stepId=step2, result={"__type":"com.aws.iot.tg.core.ast.tdm.PropertyValue",
"property": {
"__type":"com.aws.iot.tg.core.ast.tdm.Property","propertyAlias":"_", "propertyTypeUrn":
{"__type":"com.aws.iot.tg.core.ast.tdm.URN","id":"urn:tdm:aws:Property:Int16"},"value":0}}
```

The log file also contains entries that provide information about the requests being sent to the devices and the responses from the devices. The requests and responses are represented as JSON payloads. The following example shows a successful request from AWS IoT Things Graph to a Modbus device, and the response sent by the device. If the device uses the MQTT protocol, the request and response entries would show the MQTT topics to which the devices are subscribed. (AWS IoT Things Graph creates these subscriptions when you deploy the flow.)

```
[INFO]-Publishing message on topic modbus/adapter/request with payload {"request":
{"address":4,"count":1,"device":1,"operation":"ReadHoldingRegistersRequest"},"id":"44685cb0-98f0-4f81-8bbd-7c8ac5494d6b"
[2019-03-12T21:12:09.473Z][INFO]-Got work item with invocation id [7601c5b0-3482-426b-4e81-8e81c96a8486]
[2019-03-12T21:12:09.473Z][INFO]-v0.8.7
[2019-03-12T21:12:09.473Z][INFO]-Lambda input: {response={status=success, device=1, operation=ReadHoldingRegistersRequest, payload={function_code=3, registers=[0], address=4}}, id=937c3d67-36db-4403-9692-d2e8c832e70f}
{"status":"success","device":1,"operation":"ReadHoldingRegistersRequest"","payload":
{"function_code":3,"registers":0,"address":4}},"id":"937c3d67-36db-4403-9692-d2e8c832e70f"
```

These request and response entries are where you can verify that your devices are interacting correctly in the flow. You can use these logs to determine whether any of the devices in your flow are having connectivity issues, or whether their certificates are set up correctly. In those cases, you'll see in the log file that certain devices in the flow aren't returning responses.

For more information about troubleshooting AWS IoT Things Graph flows, see Troubleshooting AWS IoT Things Graph.
Troubleshoot issues with AWS IoT Things Graph

Use the following information to help troubleshoot issues in AWS IoT Things Graph.

If your deployment target is an AWS IoT Greengrass group, you might run into issues that relate directly to AWS IoT Greengrass. For AWS IoT Greengrass troubleshooting information, see Troubleshooting AWS IoT Greengrass. You also need to configure your AWS IoT Greengrass group to work with AWS IoT Things Graph. For information about setting up an environment for deploying AWS IoT Things Graph workflows (flows), see Setting up your environment.

General deployment issues with AWS IoT Things Graph

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the AWS IoT Things Graph console, you see Deployed in target in the Status column for your system instance, but your flow isn’t running.</td>
<td>The deployment to your AWS IoT Greengrass might have failed.</td>
</tr>
<tr>
<td></td>
<td>• On the AWS IoT console (<a href="https://console.aws.amazon.com/iot/">https://console.aws.amazon.com/iot/</a>), choose the AWS IoT Greengrass tab. Choose the AWS IoT Greengrass group you’re using for AWS IoT Things Graph flow deployments. Check the status of your latest deployment. Issues in your AWS IoT Greengrass group might be related to a failed deployment to AWS IoT Greengrass. For AWS IoT Greengrass troubleshooting information, see Troubleshooting AWS IoT Greengrass.</td>
</tr>
<tr>
<td></td>
<td>• Or the failed deployment might be related to an AWS IoT Greengrass group or core that isn’t set up for AWS IoT Things Graph. For example, you might need to install AWS IoT Greengrass version 1.7 or later. For information about setting up an environment for deploying AWS IoT Things Graph flows, see Setting up your environment.</td>
</tr>
<tr>
<td></td>
<td>When you deploy a flow for the first time, a file named thingsgraph.db appears in the thingsgraph directory that you created at the root of your AWS IoT Greengrass core device. You can verify that the flow is running by checking the ThingsGraph.log file in the /greengrass/ggc/var/log/user/region/ThingsGraph directory.</td>
</tr>
</tbody>
</table>
### Modbus flow deployment issues with AWS IoT Things Graph

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your flow contains a service that exposes the capability of an AWS Lambda function. (For an example that contains these kinds of services, see Creating a flow with Lambda functions.) The AWS IoT Greengrass deployment fails with an error message indicating that the Lambda function doesn’t exist in your AWS IoT Greengrass group.</td>
<td>On the AWS IoT console (<a href="https://console.aws.amazon.com/iot/">https://console.aws.amazon.com/iot/</a>), go to the AWS IoT Greengrass tab. Choose the AWS IoT Greengrass group you're using for AWS IoT Things Graph flow deployments, and then choose the Lambdas tab. Verify that you've added the Lambda function used in your service to the group. Also verify that you're using the correct version of the function. For example, the AWS IoT Things Graph services used in Creating a flow with Lambda functions require you to add version 1 of each function to your group.</td>
</tr>
<tr>
<td>Your flow contains a Modbus device. The flow appears to have deployed, but it isn't running.</td>
<td>Make sure that you've installed the Modbus-RTU Protocol Adapter.</td>
</tr>
<tr>
<td>Your flow contains a Modbus device. The AWS IoT Greengrass deployment fails with this error: unable to load the group file downloaded file Path path doesn’t exist.</td>
<td>Make sure that you've added your device to your AWS IoT Greengrass group as a local resource. For instructions about how to add local resources to a group, see Access local resources with Lambda functions. Because AWS IoT Things Graph runs as a Lambda function on your core device, you don't have to add a Lambda function. AWS IoT Things Graph uses this local resource. Also make sure that the source path for your device (which represents the physical Modbus serial port) is correct on the AWS IoT Greengrass tab on the AWS IoT console (<a href="https://console.aws.amazon.com/iot/">https://console.aws.amazon.com/iot/</a>).</td>
</tr>
</tbody>
</table>
AWS IoT Things Graph Data Model
reference

The AWS IoT Things Graph Data Model (TDM) is a declarative framework designed for representing IoT concepts, devices, and workflows. It also provides semantic mappings between multiple representations of IoT concepts, such as color, brightness, and temperature. GraphQL provides the language and syntax for expressing TDM concepts.

Data models and mappings created in this framework enable automation engineers to build workflows consisting of devices from multiple manufacturers. Because the IoT devices and their properties are represented and mapped in TDM, devices can interoperate and communicate with each other without any transformations or low-level code on the devices. For example, TDM can map equivalent units of measure across devices that use different scales, such as Fahrenheit and Celsius.

Additionally, the TDM framework enables IoT automation engineers to build workflows consisting of device models created in TDM. Physical devices can be assigned to device models, and their interactions in a workflow can be defined in TDM. Engineers can then deploy multiple instances of a workflow using different sets of concrete devices in each deployment. Engineers can also swap a given device from one manufacturer with a device from another manufacturer because both devices and their properties are represented in the same device model.

TDM consists of a set of core constructs, including and most importantly PropertyTypes. TDM also consists of a set of specialized constructs that support IoT-specific constructs, including State, Device, and Capability.

The following sections describe both the core and the IoT-specific concepts of TDM. They also describe how to use GraphQL syntax to create and extend TDM constructs.

Topics

- What is the AWS IoT Things Graph Data Model? (p. 213)
- AWS IoT Things Graph Data Model and GraphQL (p. 216)
- AWS IoT Things Graph Data Model core constructs (p. 220)
- IoT domain constructs (p. 232)

What is the AWS IoT Things Graph Data Model?

The AWS IoT Things Graph Data Model (TDM) is a feature of AWS IoT Things Graph that enables users to create abstract representations of IoT devices and concepts. A concept is a virtual representation of a real-world measurement or quantity. For example, temperature and color are concepts. In TDM, concepts are called Properties and are important building blocks. Further, a key feature for enabling interoperability among devices is the Mapping. A mapping expresses a conversion between different representations of a property. This makes it possible for disparate devices from multiple manufacturers to communicate with each other.

The following snippet from an example discussed in Mapping shows how this works. It creates a simple translation of one way of representing the on/off property of a device (an enum) into an alternative way of representing the same concept.

```graphql
query OnOffEnum_to_OnOffInt @mapping(id: "urn:tdm:aws:Mapping:OnOffEnum_to_OnOffInt") {

}
```
In the following topics, we discuss the syntax at greater length. For now we can focus just on the \texttt{from} and the \texttt{to} properties. A lot of devices can have on/off states, so this property isn't nested deeply in the property hierarchy, but it's a basic, granular concept. Two representations of it can be precisely translated in both directions, and this mapping can be used across all devices that represent the on/off state in one of these two ways.

**GraphQL syntax**

TDM is expressed with GraphQL syntax. TDM constructs are compliant with the GraphQL standard, but TDM uses GraphQL in a distinct and highly specialized way.

TDM doesn't use GraphQL as a query language or as a server runtime for executing queries. Instead, TDM uses GraphQL as a tool for creating a type system consisting of IoT devices and data concepts. GraphQL provides a powerful and concise way of defining the types and structures common in IoT. Since Graphs are useful tools for modeling real-world objects (such as IoT devices) and GraphQL models things as graphs, its syntax is a convenient vehicle for modeling the IoT conceptual space.

You can look at the [GraphQL documentation](https://graphql.org/) for a thorough understanding of GraphQL and what you can do with it. However, to understand how it's used in TDM, you need to understand only a handful of core concepts.

**Types**

GraphQL uses types to define the structure of objects in a schema. The given type of an object must define the object's structure. For example, an employee address book would need to define an object type for \texttt{Employee}.

The following example shows how you might define an \texttt{Employee} in GraphQL.

```graphql
type Employee {
  id: ID!
  firstname: String!
  middlename: String
  lastname: String!
  location: Building!
  phonenumbers: [PhoneNumber!]!
}
```

- \texttt{Employee} – A GraphQL object type. This type has six fields.
- \texttt{id, firstname, middlename, lastname, location, phonenumbers} – Fields on the \texttt{Employee} object type. Any object of type \texttt{Employee} must contain these fields.
- \texttt{String} – A built-in data type. The exclamation point (!) means that the field can't be null.
- \texttt{Building} – Another GraphQL object type. Its structure must also be defined in the schema.
- \texttt{PhoneNumber} – Another GraphQL object type. Its structure must also be defined in the schema. The brackets mean that this is an array of \texttt{PhoneNumber} objects. The exclamation point inside the brackets means that no object in the array can be null. The exclamation point outside the brackets means that the array itself can't be null. Because an empty array is technically not null, and an empty array contains no elements, an empty array would be valid according to this definition.

**Queries**
Queries in GraphQL are typically used to execute queries on objects within a GraphQL schema. However, TDM uses queries in much the same way as types. A query in TDM is simply a way of defining objects that have a greater degree of complexity than the type syntax supports. The original purpose of GraphQL queries is to request fields on objects. Because a GraphQL schema can contain nested types (as in the Employee example, which contains both Building and PhoneNumber objects), queries on GraphQL objects can contain nested structures. A query requesting the name and address of a particular employee, for example, could look like the following example.

```graphql
query {
  Employee(id: 1000) {
    name
    building {
      name
      address {
        streetAddress
        zipcode
      }
    }
  }
}
```

In TDM, GraphQL queries aren't used for executing queries on objects that are defined in a GraphQL schema. In TDM, queries are used for representing complex concepts, such as Mappings.

### Directives

A directive is an instruction attached to a field that begins with the `@` symbol. Directives in GraphQL are generally attached to query fields and are meant to manipulate the results of a query in any way defined by the server runtime. The GraphQL standard currently defines only two directives: `@include(if: Boolean expression)` and `@skip(if: Boolean expression)`. You can use these directives to specify the conditions under which a field gets included in a result set. TDM uses directives in a very different way.

In TDM, directives play a fundamental role in defining IoT concepts. Instead of defining query behavior, TDM directives are conceptually the equivalent of statements and type declarations.

In the following example, the `@enumType` directive tells AWS IoT Things Graph to create a simple ON/OFF enum. Notice that this enum is used in the preceding mapping snippet.

```graphql
enum onOff @enumType(id: 'urn:tdm:aws:enum:onOff') {
  ON @enumValue(value: 0),
  OFF @enumValue(value: 1)
}
```

The directive follows the `enum` GraphQL object type specification and name. In this context, it means "create an enum with the following properties and structure." The directive takes one or more arguments that specify the properties of the object to create. In this case, the enum type requires only a TDM URN.

As in regular GraphQL objects, the fields inside the curly braces define the structure of the object. The enum contains only two fields, ON and OFF. These fields also have directives attached to them. In this case, the `@enumValue` directive tells AWS IoT Things Graph to create enum values with the values specified by the arguments inside the parentheses: `value: 0` and `value: 1`.

TDM combines these three GraphQL concepts—types, queries, and directives—into a rich and expressive tool for defining IoT applications.
Now that you have a general understanding of these concepts, see AWS IoT Things Graph Data Model and GraphQL for more information about how TDM works.

AWS IoT Things Graph Data Model and GraphQL

The AWS IoT Things Graph Data Model (TDM) uses GraphQL syntax to describe the structure and logic of constructs and concepts that are specific to the IoT domain. In particular, it uses two specific GraphQL concepts—types and queries—in combination with GraphQL directives to represent constructs in the IoT domain.

TDM doesn’t use other GraphQL concepts, such as mutations and subscriptions, and it doesn’t interact with a GraphQL server or runtime. Although it uses GraphQL in a way that is different from its original purposes, TDM is fully compatible with GraphQL. It extends GraphQL in a way that is analogous to the way in which the C++ programming language extends C.

Instead of querying data from a GraphQL server, TDM enables you to create models of IoT devices and stateful web applications, and define how they interact within an IoT system.

For more information about GraphQL and how to use it more generically, see the GraphQL documentation.

The following sections describe how specific GraphQL concepts are implemented in TDM, and how the TDM constructs fit together at the highest level.

**Type**

TDM uses GraphQL types to describe IoT concepts that can be represented as data structures without requiring complex payloads or logic. Types in TDM are the basic building blocks of an IoT system. The TDM constructs that are described as types are Properties, States, Events, Actions, Capabilities, and Systems.

The following example uses a GraphQL type to create a property named `imageUri`. The example demonstrates additional TDM concepts that are discussed in later sections. This property is a relatively simple structure with two arguments. It also contains no payload.

*Note*

TDM keywords and variables are case insensitive.

```
type imageUri @propertyType(id: "urn:tdm:aws:property:Uri", dataType: STRING){ignore:void}
```

The `@propertyType` directive (discussed in the section called “Using directives” (p. 217)) specifies this as a TDM Property. The first argument assigns a unique identifier according to the TDM URN scheme. The second argument specifies the data type of this property as a string.

The `ignore:void` name-value pair between the braces indicates that this type contains no payload. GraphQL syntax requires a trailing brace block with one or more field definitions. This name-value pair works around that requirement in cases where no payload is required.

**Query**

TDM uses GraphQL queries to describe IoT concepts that are represented by logic and/or complex data structures that can't be described by the syntax of GraphQL types. TDM queries contain structured data
with logic that must be executed when IoT devices and services are interacting with each other. The TDM constructs that are described as queries are Mappings, Devices, Services, and Workflows.

All of these constructs represent dynamic IoT interactions or direct participants in these interactions. Mappings translate data received from one device into standards and formats that other devices can understand. Workflows describe how devices in an IoT system interact in real time. Triggers start workflows, and devices interact with each other within workflows. Additionally, all of these constructs require relatively complex syntax to describe them.

The following example describes a device named myMotionSensor. This device inherits from the MotionSensor device model and implements a capability associated with that device model. The capability contains a state and an event. The state, in turn, contains properties.

The specific TDM concepts demonstrated in this example are discussed in sections that follow in this guide. The example demonstrates the general shape of a TDM query and the complexity that it supports.

```
query myMotionSensor @device(id: "urn:tdm:aws:device:MotionSensor", deviceModel: "urn:tdm:aws:deviceModel:MotionSensor") {
  MQTT {
    MotionSensorCap(id: "urn:tdm:aws:capability:MotionSensorCap") {
      state {
        isMotionDetected(name:"lastMotionDetected", property:"urn:tdm:aws:property:MotionDetected")
      }
      event(name: "motionDetected") {
        Subscribe(topic: "motionsensor/motion") {
          responsepayload(property: "urn:tdm:{{NAMESPACE}}:Property:MotionDetected")
        }
      }
    }
  }
}
```

The @device directive specifies this as a TDM Device. As in the property definition, the first argument assigns a unique identifier according to the TDM URN scheme. The second argument contains another TDM URN that specifies the device model from which this device inherits its capability. The definition then implements the device model's capability by including a State and an Event. A capability can also include an Action. If the device model's capability includes an Action, the device must implement it as well.

**Using directives**

TDM uses GraphQL directives extensively for implementing concepts that are specific to the IoT domain. Directives can be attached to any field to insert domain-specific information. The preceding examples demonstrate how directives are used to specify a specific TDM construct (the propertyType and device directives). In TDM, directives play the role of statements. When you see a directive in a type or query definition, it means "create an instance of this type".

TDM supports two kinds of directives: definition and usage. You use definition directives to create instances of a type of construct. For example, the @propertyType directive tells AWS IoT Things Graph to create a property. The following example creates a property named MotionDetected by using the @propertyType definition directive.

```
type MotionDetected @propertyType(id: "urn:tdm:aws:property:MotionDetected", dataType: Boolean){ignore:void}
```
The @propertyType directive tells AWS IoT Things Graph to create a property. The first argument contains a TDM URN that uniquely identifies the new property. This URN must be in the property branch of the TDM URN scheme. The URN scheme and the directive must match. The second argument specifies the property's data type, which in this case is the built-in primitive Boolean. The ignore: void name-value pair between the braces specifies an empty payload.

A usage directive is a specific instance of a type. For example, after you create the MotionDetected property type, you can use it to construct a State by using the @property usage directive, as in the following example.

```graphql
type MotionSensorState @stateType(id: "urn:tdm:aws:state:MotionSensorState") {
  isMotionDetected : MotionDetected @property(id:"urn:tdm:aws:property:MotionDetected")
}
```

The @stateType directive tells AWS IoT Things Graph to create a state definition. The urn:tdm:aws:property:MotionSensorState argument uniquely identifies the new state and types it as a state by placing it under the state branch of the TDM URN scheme. The URN scheme and the directive must match. The first piece of the payload between the braces specifies the name of the state as it will be used when a device implements the state. The pieces of the payload that follow the first colon specify the property that the state contains by name, by usage directive, and by URN. The @property directive in this example means "use the property specified by this URN".

Notice that the isMotionDetected state name appears as part of the state implementation in the preceding device definition example.

```graphql
state {
  isMotionDetected(name:"lastMotionDetected",
                   property:"urn:tdm:aws:property:MotionDetected")
}
```

Directives and URNs provide a way of implementing type safety in GraphQL structures. When you create a TDM construct by constructing a GraphQL type or query, you must create a unique URN that places the construct inside a conceptual hierarchy that matches the type specified by the directive. When you use an instance of the type, you must supply a URN that already exists and that sits at a location in the conceptual hierarchy that matches the type specified by the directive.

The following table contains the usage and definition directives in TDM.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>@actionType</td>
<td>@action</td>
</tr>
<tr>
<td>@capabilityType</td>
<td>@capability</td>
</tr>
<tr>
<td>@deployment</td>
<td></td>
</tr>
<tr>
<td>@deviceModel</td>
<td>@device</td>
</tr>
<tr>
<td>@enumType</td>
<td>@enum</td>
</tr>
<tr>
<td>@enumValue</td>
<td></td>
</tr>
</tbody>
</table>
Creating a device with GraphQL

To construct a complete device, you need to use the following constructs and associated directives.

- **Property** (type @propertyType): The attributes of a device, such as the color and brightness of a light bulb and the click time of a camera.
- **Device Model** (type @deviceModel): An abstraction of a device that contains a capability. For example, you use a device model definition to create abstract definitions of things like light bulbs and cameras.
- **Device** (query @device): A device type that inherits from the device model and implements the device model's capability. Cameras and light bulb models produced by specific manufacturers would be defined as devices.
- **Capability** (type @capabilityType): The actions, events, and state associated with the device. A light bulb's capability would include its brightness (part of its state), the action of changing its brightness, and a power on/off event. A camera's capability would include its burst delay interval (part of its state), the action of capturing an image, and the camera click event.
- **Action** (type @actionType): The device's ability to perform a function. A light bulb can change its brightness, and a camera can capture an image. Both are examples of actions.
- **Event** (type @eventType): The device's ability to communicate externally with other devices and services. A light bulb can send a notification that it has been turned on or off, and a camera can send a notification that it has been clicked. Both are examples of events.
- **State** (type @stateType): The properties that describe the device at a specific point in time. A light bulb has properties like brightness and color. A camera has properties like image URI (for a specific image that it has captured) and click time (for the time when an image was captured). A state is a collection of these properties.
- **Mapping** (query @mapping): Symmetrical, forward, and/or reverse transformations of properties and measurements that enable different device types to understand each other. A mapping can transform an enum-based brightness rating into a rating based on a numerical range. It can also convert the burst delay property of a camera from milliseconds into seconds.

The following diagram shows how all of these constructs fit together in a device model.
AWS IoT Things Graph Data Model core constructs

This part of the guide describes the core constructs, properties, and entities that the AWS IoT Things Graph Data Model (TDM) supports, and the GraphQL syntax for creating and editing them.

TDM has a set of built-in data types that support standard primitive data types, and the more complex data types that are specific to the IoT domain. TDM also uses a URN scheme for identifying all types in the system and organizing them into hierarchies.

Topics
- Built-in data types (p. 220)
- URN ID scheme (p. 221)
- Properties (p. 223)
- Enum types (p. 225)
- Expressions (p. 225)

Built-in data types

TDM provides built-in support for the standard primitive data types included in the following table.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>Unicode string</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
<td>Boolean true or false</td>
</tr>
<tr>
<td>int8</td>
<td></td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>int16</td>
<td></td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>int32</td>
<td></td>
<td>32-bit signed integer</td>
</tr>
</tbody>
</table>
## Data type | Alias | Description
--- | --- | ---
int64 |  | 64-bit signed integer
uint8 | byte | 8-bit unsigned integer
uint16 |  | 16-bit unsigned integer
uint32 |  | 32-bit unsigned integer
uint64 |  | 64-bit unsigned integer
float32 | float | IEEE 754 single-precision floating point
float64 | double | IEEE 754 double-precision floating point
urn |  | Uniform Resource Name (URN): adaptation of 2017 RFC 8141
arn |  | Amazon Resource Name (ARN)
uri |  | Uniform Resource Identifier (URI): a resource path
datetime |  | ISO 8601
date |  | Date component of datetime
duration |  | Duration expressed as time; e.g., 30 minutes, 20 seconds

## URN ID scheme

TDM URNs provide a hierarchical way of typing devices, properties, and entities. They provide organizational information (such as physical location) about a type and semantic information (such as color or the URI for an image).

The TDM URN type ID consists of three components:

- Hierarchical namespace
- Metatype
- Hierarchical semantic type path (delimited with '/') that is an instance of the metatype

```
name of urn-scheme namespace semantic-type-path delimited with '/'
```

```
urn:TDM:namespace:meta-type:sem-type-path
```

urn namespace identifier. ("TDM" is urn namespace) semantic metatype
urn
Identifier that is fixed and required by the URN scheme.

TDM
Fixed identifier that is the URN namespace for the AWS IoT Things Graph Data Model.

namespace
Identifier provided by the user. There are two kinds of namespaces: public and private. The public namespace contains the built-in entities (such as primitive properties) supported by AWS IoT Things Graph. A user's private namespace contains entities, workflows, and other AWS IoT Things Graph entities that the user creates. Private namespaces are versioned, and they track and synchronize with the public namespace. For more information about namespaces, see the definition in the Glossary.

metatype
A TDM data metatype such as device, action, event, and property.

sem-type-path
The hierarchical path that specifies an instance of the metatype. When the metatype is Action or Event, the sem-type-path values specify a capability and an action or event that is an instance of the capability, as in the following example.

<table>
<thead>
<tr>
<th>TDM metatype</th>
<th>Example TDM URN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>urn:tdm:aws:enum:actionStatus</td>
</tr>
<tr>
<td>Category</td>
<td>urn:tdm:aws:category:color</td>
</tr>
<tr>
<td>Category (subcategory)</td>
<td>urn:tdm:aws:category:color/red</td>
</tr>
<tr>
<td>Device</td>
<td>urn:tdm:aws:device:flashlight</td>
</tr>
<tr>
<td>Property</td>
<td>urn:tdm:aws:property:switch</td>
</tr>
<tr>
<td>Property (sub-property)</td>
<td>urn:tdm:aws:property:switch/powerOnOff</td>
</tr>
<tr>
<td>State</td>
<td>urn:tdm:aws:state:brightness</td>
</tr>
<tr>
<td>Capability</td>
<td>urn:tdm:aws:capability:camera</td>
</tr>
<tr>
<td>Action</td>
<td>urn:tdm:aws:action:camera/capture</td>
</tr>
<tr>
<td>Event</td>
<td>urn:tdm:aws:event:camera/clicked</td>
</tr>
<tr>
<td>Annotation</td>
<td>urn:tdm:aws:annotation:cameraUsage</td>
</tr>
<tr>
<td>Mapping</td>
<td>urn:tdm:aws:mapping:brightnessRangeEnumToNumber</td>
</tr>
<tr>
<td>Workflow</td>
<td>urn:tdm:aws:flow:simpleHomeSecurity</td>
</tr>
</tbody>
</table>
Properties

The PropertyType construct is used to define the attributes of a device. For example, a device named Light Bulb might have properties like PowerState, Brightness, and Color.

The following example shows how to create a PropertyType using GraphQL.

```graphql
type imageProperty @propertyType(id:"urn:tdm:aws:property:image/imageUri", 
   dataType:"urn:tdm:aws:property:Uri") { ignore:void }
```

Note
The ignore:void name-value pair specifies an empty body payload.

PropertyType facets

The PropertyType construct is defined by property facets. There are two kinds of facets:

- **Definition** - These facets help create or define a PropertyType. Example definition facets include Name, DataType, Accuracy, and Uom. You can specify the Uom and Accuracy facets with instance values. For example, a PropertyType named Length that has a Uom facet can be assigned an instance value of 10 inches or 10 feet.

- **Usage** - These facets help customize a PropertyType within a specific scope. Example usage facets are calculated values like Condition and Expr.

TDM facets are case insensitive.

Definition facets

**SemTypeId**
Required, multivalued. A URN or URNs that represent a property's semantic type.

The semantic type hierarchical path asserts an "is-a" relationship contract between parents and children in the path. This relationship establishes semantic equivalence and doesn't imply structural equivalence. The subtype is a semantically equivalent alternate of its parent type for the purpose of interoperability in an IoT workflow. For example, a brightness property expressed by a range of numbers between 1 and 100 is semantically equivalent to a brightness property expressed as an enum.

Example: urn:tdm:aws:property:length

**DataType**
Required. A value from the built-in enum DataType. See Built-in data types for the possible values for this facet.

You can also limit the range of an integer property type by using the minValue and maxValue attributes, as in the following example. The property type definition creates an on/off property that is represented as a 0 for off and 1 for on.

```graphql
type OnOffInt @propertyType(id:'urn:tdm:aws:Property:OnOffInt', 
   dataType:int8, 
   minValue:0, maxValue:1) {}
ignore: void
}

**Description**

Optional. A string that describes the property.

**InstanceOf**

A URN that specifies a complex property's type. This value is required whenever the property's `DataType` value is not one of the built-in primitive data types.

**Uom**

Optional. A value from the built-in enum under the semantic path at this URN: `urn:tdm:aws:enum:uom`. This value specifies the unit of measure associated with the property's value.

*Example:* `urn:tdm:aws:uom:base/mass`

**Accuracy**

Optional. A float32 value that describes the closeness of a measurement's value to the true value. This value can be expressed as either a +/- absolute value or a percentile value. TDM doesn't process this value.

**IsAbstract**

Optional. A Boolean value that specifies whether the `PropertyType` can be used in a concrete type. The default value is `false`. An abstract `PropertyType` is usually the root of a property hierarchy.

**IsGlobalId**

Optional. A Boolean that specifies whether the value of a property is unique across all instances of the type that contains the property. The default value is `false`. It applies only to non-nullable identifier properties. If the value is `true`, the instances of the type that contains the property can be used by reference. A `DeviceCatalog` is an example of the kind of type that would have this facet set to `true`.

**MinLength, MaxLength**

Optional. Number values that specify in bytes the minimum and maximum length of a string property. The default value of `MinLength` is 0. The default value of `MaxLength` is 4096. When both values are the same, the property has a fixed length.

**MinValue, MaxValue**

Optional. Number values that specify the minimum and maximum values of a numeric property. The numeric property can be of any type of int, uint, and float.

**Usage facets**

**IsRequired**

Optional. A Boolean value that specifies whether a property is required for every type that contains the property. The default value is `false`.

**Default**

Applicable only to properties for which the `IsRequired` facet has been set to `true`. A variant-typed facet that specifies the default value of a property.
Condition

Optional. A Boolean expression that evaluates other properties of the type that contains the
property. If the expression evaluates to true, the property value is included. If the expression
evaluates to false, the property is excluded (either null or default, as applicable). A
PropertyType with this facet is read-only.

Expr

Optional. An expression that converts a standard property into a computed property. A
PropertyType with this facet is read-only.

IsReadOnly

Optional. A Boolean facet that specifies whether the property is read-only. The default value is
false.

Multiplicity

Optional. An enum facet that specifies the number of values that a property can have. The default
value is Multiplicity.Multivalued. A value of Multiplicity.Singlevalued limits the
number of values of a property to 1.

Enum types

The EnumType construct defines an enumeration that associates unique identifiers to integer values
within a specified scope.

Examples

```
enum lowMediumHigh @enumType(id:'urn:tdm:aws:enum:lowMediumHigh')  {
    Low @enumValue(value:0),
    Medium @enumValue(value:1),
    High @enumValue(value:2)
}
enum trueFalse @enumType(id:'urn:tdm:aws:enum:trueFalse')  {
    TRUE @enumValue(value:0),
    FALSE @enumValue(value:1)
}
```

Expressions

This topic describes the kinds of expressions that you can write in AWS IoT Things Graph entities. It also
explains which AWS IoT Things Graph entities can use expressions and the types and scopes of variables
that you can use in each entity.

How to write AWS IoT Things Graph expressions

The AWS IoT Things Graph Data Model (TDM) provides several ways to specify values whenever you
define or use TDM constructs. You can often specify literal values, but on many occasions you might need
to evaluate information before specifying a particular value.

Workflows need to manipulate and evaluate data before determining what actions to take. For example,
certain steps can't be taken unless a given condition is met. In that case, a Boolean expression might be
appropriate.
Mappings use expressions to transform information from one device into representations that other devices can understand. A service or device definition might require end user input, and this input can be represented as variables in TDM expressions.

The following list contains the types of expressions that TDM supports. Variables can be used in expressions. Each expression type is followed by a set of examples.

**Literal**

Expressions that are static instances of the built-in primitive data types.

- `'hello'`  # String literal
- `123`  # Int literal
- `[1,2,3]`  # Array literal
- `10L`  # Long literal
- `10UL`  # Unsigned long literal

**Predicate**

Expressions that evaluate to true or false.

- `True`
- `False`
- `10 < 13`
- `#aValue > #bvalue`
- `(10 + 11) == 21`

**Path**

Expression that enables navigation through a complex object. TDM supports path expressions that use both the forward slash (/) and dot (.) notations.

- `image.value`
- `image/value`
- `users[name == 'rob'].height`

**Arithmetic**

Expression that results in a numeric value. The values used in an arithmetic expression can be other kinds of expressions.

- `1 + 2`
- `100UL - 3.4L`
- `'abc' + 1 + 2`
- `(3 / 4) / (5 / 4)`
- `users[name == 'rob'].height * 0.9`

**Ternary**

Conditional expressions that use the ? operator: `condition ? expr1 : expr2`. 
Expressions

$aValue < $bvalue ? 'This is returned if $aValue is less than $bValue' : 'This is returned if $aValue is not less than $bValue'

IsNull

Expression that uses two $ symbols to determine whether the value to the left of the operator is null. If the value to the left of the operator is null, the value to the right is returned. If the value to the left of the operator is not null, the value to the left is returned.

$aValue ?? 'This string is returned if $aValue is null. $aValue is returned if it isn't null.'

Function calls

TDM supports a set of utility functions. These functions wrap the utility functions in java.lang.math and java.lang.string. All of these begin with tdm.lib. Array type parameters aren't supported.

tdm.lib.Math.log(2.71728) # Evaluates to 0.999999989

InstanceOf

TDM supports the use of instance methods on primitive data types. These are similar to instance methods in Java, such as String.substring() and Integer.doubleValue().

((string) 'a' + 'b').substring(1)

Macro

TDM supports interpolation of values into a string by wrapping expressions inside ¦.}

macro(This is a string with ¦3 + 4¦ words) # Evaluates to 'This is a string with 7 words'
macro({"name":"\$\'jeff' + ' ' + 'b'"}, "age": $\{100 % 52\}) # Evaluates to '{"name":"jeff b", "age": 48'}

You can use expressions in Devices, Services, Triggers, Workflows (Flows), and Mappings. constructs. (See the section called "Using conditional logic in choice nodes" (p. 120), the section called "Workflow" (p. 244), and the section called "Mapping" (p. 233) for examples.) The examples in those sections demonstrate how to use expressions to evaluate and manipulate data supplied by devices and services.

Working with variables in expressions

Expressions and variables are available in triggers, devices, services, workflows, and mappings. The variables and parameters that are available to expressions depend on the entities in which you include them. This section explains which AWS IoT Things Graph entities can include expressions and the variable types and scopes that are available to each entity.
Triggers

Expressions in triggers have access to the deviceId bindings inside the deployment parameters and to values in the trigger device output. The following example from the section called “Creating a flow with devices by using the AWS CLI” (p. 18) shows how you can use expressions in a trigger.

```
query Room215 @deployment(id: "urn:tdm:REGION/ACCOUNT_ID/default:Deployment:Room215", systemId: "urn:tdm:REGION/ACCOUNT_ID/default:System:SecurityFlow") {
  motionSensor(deviceId: "MotionSensorName")
  screen(deviceId: "ScreenName")
  camera(deviceId: "CameraName")
  triggers {MotionEventTrigger(description: "a trigger") {
    condition(expr: "devices[name == 'motionSensor'].events[name == 'StateChanged'].lastEvent")
    action(expr: "ThingsGraph.startFlow('SecurityFlow', bindings[name == 'camera'].deviceId, bindings[name == 'screen'].deviceId)")
  }
}
```

In triggers, you wrap expressions in quotation marks and use them as values of the expr: field. You access the trigger device output by specifying the device name (name = 'deviceName'). You then use the .events path to specify the triggering event that generates the output (name == 'eventName').

The preceding code sample does the following:

- The first expression, in the condition section, uses output from the motion sensor device that triggers the flow (motionSensor.events['StateChanged'].lastEvent).
- The second expression, in the action section, uses the bindings defined in the parameters section of the deployment directive (motionSensor, screen, and camera).
- The action section also uses the startFlow method available from the ThingsGraph global. You can use this function only in the action section of a trigger.

You must identify the name of the associated deployment parameter for each deviceId binding.

**Workflows (Flows)**

Flows can include expressions inside steps and choice node rules inside those steps. Expressions in flows have access to device and service outputs and to variables defined in the flow. Expressions also have access to global values that are stored in the systemRuntime namespace.

The following list contains the values that are available in this namespace:

- systemRuntime.awsAccountId
- systemRuntime.awsRegion

The following example choice node shows how you can use expressions in a flow.

```
ChoiceActivity {
  rule(expr: "#${getHourResult > 14}", name: "Night") {
    setEvent(name: "isNight")
  }
  rule(expr: "#${getHourResult <= 13}", name: "Day") {
    setEvent(name: "isDay")
  }
  default {
    setEvent(name: "isNight")
  }
}
```
In flows, you wrap expressions in ${} blocks. In choice node rules, you also use them as values of the expr: field.

The getHourResult value can be an output from a service or a device, such as a clock, that performs an earlier step inside the flow.

Flows use expressions to retrieve values from service and device outputs and make them inputs for devices and services later in the flow. The following example shows how you can use expressions to retrieve an output value from an AWS Lambda service and use it as an input to another service. (This example is adapted from the section called "Creating a flow with Lambda functions by using the AWS CLI" (p. 46).)

```plaintext
step(name:"getS3ObjectAsStringStep", outEvent:["TextReady"]){
    WebserviceActivity(webservice:"urn:tdm:NAMESPACE:Service:getS3Lambda", out:"textDocument") {
        getS3ObjectAsString(key: "HelloWorld.txt")
    }
}
step(name:"wordCounts", inEvent:["TextReady"], outEvent:["DoneProcessing"]){
    WebserviceActivity(webservice:"urn:tdm:NAMESPACE:Service:wordCountLambda", out:"wordCounts") {
        wordCount(message: "${textDocument.message}")
    }
}
step(name:"save", inEvent:["DoneProcessing"], outEvent:["lambdaDone"]){
    WebserviceActivity(webservice:"urn:tdm:NAMESPACE:Service:saveResponseLambda") {
        save(response: "${wordCounts}")
    }
}
```

The preceding example does the following:

- The getS3Lambda service outputs textDocument, and the wordCountLambda service uses the ${textDocument.message} expression to retrieve the message value from the textDocument object.
- The saveResponseLambda service takes the wordCounts output from the wordCountLambda service and uses the ${wordCounts} expression to retrieve the word count value.

**Devices and services**

Devices and services have access to the following values that are stored in the systemRuntime namespace:

- systemRuntime.deviceId
- systemRuntime.awsAccountId
- systemRuntime.awsRegion

In devices and services, expressions have access to different kinds of parameters depending on where you use them:

- Expressions in devices have access to device and action parameters.
- Expressions in services have access to service and action parameters.
Expressions in the Request and Response topics used by MQTT devices have access only to device parameters.

You can use expressions only as parameter values (after the value: field).

The following example shows how to use expressions in a device.

```graphql
query MockDeviceLambdaB @device(id: 'urn:tdm:NAMESPACE:device:MockDeviceLambdaB'
deviceModel: 'urn:tdm:NAMESPACE:deviceModel:MockDeviceLambdaB') {
  MQTT {
    Capability(id: 'urn:tdm:NAMESPACE:capability:MockDeviceLambdaB') {
      Action(name: 'doSomething') {
        params {
          param(name: 'input', property: 'urn:tdm:NAMESPACE:property:Barcode')
        }
      }
      Publish {
        Request(topic: 'LambdaB/in') {
          params {
            param(name: 'readId', property: 'urn:tdm:NAMESPACE:property:Barcode'
            value: '${input.value}')
          }
        }
      }
    }
    Event(name: 'readingEvent') {
      Subscribe(topic: 'LambdaB/event') {
        responsePayload(property: 'urn:tdm:NAMESPACE:property:Barcode')
      }
    }
  }
}
```

In services and devices, you wrap expressions in `${}` blocks. You access the values of variables inside expressions by using the .value path.

In this example, the Publish block inside the doSomething action accesses the value of the input action parameter by using the `${input.value}` expression.

The following example shows how to use expressions in a service.

```graphql
query Sqs @service(id: "urn:tdm:NAMESPACE:service:Sqs") {
  REST {
    SqsCap(id: "urn:tdm:NAMESPACE:capability:SqsCap") {
      Action(name: "createQueue") {
        params {
          param(name: 'queueName', property: 'urn:tdm:aws:property:String')
        }
      }
    }
    HttpGet {
      awsServiceName: "sqs") {
        headerParams {
          param(name: "Accept", property:
"urn:tdm:aws:property:String", value: "application/json")
          param(name: "Content-Type", property:
"urn:tdm:aws:property:String", value: "application/json")
        }
      queryParams {
        param(name: "Action", property:
"urn:tdm:aws:property:String", value: "CreateQueue")
      }
    }
  }
}
```
Expressions

Expressions in the service use expressions in parameter values whenever the value isn't a literal string. As in the device example, expressions are wrapped inside {} blocks and access the variable values that are specified in action parameters (queueName and message). The expressions use the .value path to retrieve the variable values ${queueName.value} and ${message.value}.

Mappings

Mappings use expressions to access the from and to values specified in each mapping. The following example shows how to use expressions in a mapping.
map(expr:'s==Medium => 3'),
map(expr:'s==High => 5')
}
reverse {
map(expr:'t <= 1 => Low'),
map(expr:'t > 1 && t <= 3 => Medium'),
map(expr:'t > 3 => High')
}

In mappings, you wrap expressions in quotation marks and use them as values of the expr: field.

The brightnessEnumToInt mapping specifies $s$ and $t$ as aliases for the from and to parameters. Each forward and reverse mapping accesses the values of those parameters by using the expressions expr:'t <= 1 => Low'.

IoT domain constructs

The AWS IoT Things Graph Data Model (TDM) provides specialized constructs for IoT-specific scenarios. This section describes the constructs and their properties. It also provides GraphQL examples that show how to create these constructs.

Topics

- State (p. 232)
- Mapping (p. 233)
- Event (p. 238)
- Action (p. 238)
- Capability (p. 238)
- Device model (p. 239)
- Device (p. 239)
- Service (p. 241)
- Workflow (p. 244)
- System (p. 247)
- Trigger (p. 249)
- Deployments (p. 250)

State

The State construct is a set of properties that represent the inner state of a device at a point in time. You can also create complex properties that are composed of States.

Examples

Creates a light bulb state.

type lightBulbState @stateType(id: "urn:tdm:aws:state:lightBulb"){
  power: SwitchEnum @property(id: "urn:tdm:aws:Property:switch/powerOnOff"),
  brightness: brightnessNumber @property(id: "urn:tdm:aws:Property:brightness/brightnessNumber")
}
Creates a colored light bulb state that extends lightBulbState.

type coloredLightBulbState @stateType(id: "urn:tdm:aws:state:lightbulb/coloredLightBulb") implements lightbulbState @stateType(id: "urn:tdm:aws:state:lightbulb") {
  power: SwitchEnum @property(id: "urn:tdm:aws:property:switch/powerOnOff"),
  brightness: brightnessNumber @property(id:"urn:tdm:aws:property:brightness/brightnessNumber"),
  color: colorRGB @property(id="urn:tdm:aws:property:color/colorRGB")
}

Creates a camera state.

type cameraState @stateType(id: "urn:tdm:aws:state:camera") {
  imageUri: ImageLinkUri @property(id: "urn:tdm:aws:property:imageLink/imageLinkUri"),
  clickTime: DateTime @property(id: "urn:tdm:aws:Property:dateTime")
}

Creates an advanced camera state that extends cameraState.

type advCameraState @stateType(id:"urn:tdm:aws:state:camera/advCamera") implements cameraState @stateType(id:"urn:tdm:aws:state:camera") {
  imageUri: ImageLinkUri @property(id: "urn:tdm:aws:property:imageLink/imageLinkUri"),
  clickTime: DateTime @property(id: "urn:tdm:aws:Property:dateTime"),
  burstDelay: interval_ms @property(id: "urn:tdm:aws:property:interval/interval_ms"),
  burstclickCount: interval_ms @property(id: "urn:tdm:aws:property:uint8")
}

Creates complex properties that contain camera states.

type cameraStateProperty @propertyType(id: "urn:tdm:aws:property:cameraState" instanceOf: "urn:tdm:aws:state:camera") {ignore:void}

type advCameraStateProperty @propertyType(id: "urn:tdm:aws:property:advCameraState" instanceOf: "urn:tdm:aws:state:advCamera") {ignore:void}

type profCameraStateProperty @propertyType(id:"urn:tdm:aws:property:profCameraState" instanceOf: "urn:tdm:aws:state:profCamera") {ignore:void}

Mapping

The Mapping construct bridges differences across multiple representations of the same concept. It converts semantically equivalent data from one representation to another. A Mapping creates a single semantic view of data that originates from multiple sources.

TDM supports directional (forward and backward) and symmetrical mappings. The keywords Forward, Reverse, and Symmetric specify the mapping direction. TDM validates mappings by identifying duplications and gaps in the mapping space.
There are two kinds of mappings:

- Simple property mappings, which map primitive type values.
- Complex property mappings, which map properties of complex properties.

Mappings enable the AWS IoT Things Graph runtime to transform different representations of the equivalent properties. A relatively simple example of how mappings can work is the on/off status of a device. One possible representation of a device's on/off status is as an integer (0/1). The following example uses the built-in int data type to define this kind of on/off property.

```typescript
type OnOffInt @propertyType(id:'urn:tdm:aws:Property:OnOffInt', dataType:int8, minValue:0, maxValue:1) {
  ignore:void
}
```

The next example shows how to create an on/off enum and use it to represent the on/off property of a device.

```typescript
enum OnOff @enumType(id:'urn:tdm:aws:Enum:OnOff') {
  ON @enumValue(value:0),
  OFF @enumValue(value:1)
}
type OnOffEnum @propertyType(id:'urn:tdm:aws:Property:OnOffEnum',
  dataType:'urn:tdm:aws:Enum:OnOff') {
  ignore:void
}
```

The properties use different data types, but they represent the same concept. You can transform these representations in both directions by using the following mapping.

```typescript
query OnOffEnum_to_OnOffInt @mapping(id:'urn:tdm:aws:Mapping:OnOffEnum_to_OnOffInt',
  name:'OnOffEnum_to_OnOffInt',
  from:'urn:tdm:aws:Property:OnOffEnum s',
  to:'urn:tdm:aws:Property:OnOffInt t') {
  forward {
    map(expr:'s == ON => 1'), # If enum value is ON, map 1 from property s to property t.
    map(expr:'s == OFF => 0') # If enum value is OFF, map 0 from property s to property t.
  }
  reverse {
    map(expr:'t == 1' => ON), # If integer value is 1, map ON from property t to property s.
    map(expr:'t == 0' => OFF) # If integer value is 0, map OFF from property t to property s.
  }
}
```

This example uses the `@mapping` directive to create the OnOffEnum_to_OnOffInt under the mapping metatype. It assigns the URN of the OnOffEnum property to the `from` field and the URN of the OnOffInt property to the `to` field. A Forward mapping uses a set of expressions to transform values of the `from` property into values of the `to` property. A Reverse mapping uses a list of expressions to transform values of the `to` property into values of the `from` property.
The enum property alias is defined as s, and the integer property alias is defined as t. The Forward mapping transforms the enum property of ON into the integer property of 1, and the enum property of OFF into the integer property of 0. The Reverse mapping transforms the integer values into the enum values.

The expressions inside the braces contain the logic that transforms the values back and forth. For details about how to construct and use TDM expressions, see Expressions.

**Additional examples**

The following example creates a mapping between two simple properties that represent brightness. The example creates the following constructs:

- An enum representation of brightness
- A numerical range representation of brightness (1-100)
- Forward and Reverse mappings between the two representations

```java
# Create abstract property simpleRange
type simpleRangeProperty @propertyType(id: "urn:tdm:aws:property:simpleRange") {}

# Create concrete property simpleRangeEnum that assigns SimpleRangeEnum to
class simpleRangeProperty @propertyType(
  id:"urn:tdm:aws:property:simpleRange/simpleRangeEnum",
  description: "Describes brightness with an enum",
  dataType:"urn:tdm:aws:property:enum:SimpleRange") {}

# Create abstract property brightness
type brightnessProperty @propertyType(id: "urn:tdm:aws/property/brightness",
 isAbstract:true) {}

# Create concrete property brightnessSimpleRange that sets the possible values for the
# brightness property
type brightnessSimpleRangeProperty @propertyType(
  id:"urn:tdm:aws:property/brightness/brightnessSimpleRange",
  description: "Brightness value: low-medium-high",
  dataObject:"urn:tdm:aws:property:enum:SimpleRange")

# Create concrete property brightnessNumber that sets the possible values for the
# brightness property
type brightnessNumberProperty @propertyType(
  id:"urn:tdm:aws:property/brightness/brightnessNumber",
  description: "Brightness value from 0-100",
  dataObject:"urn:tdm:aws:property:enum:SimpleRange")

# Map Enum(low-medium-high) to integer (1-100) range representation
query brightnessSimpleRangeToBrightnessNumberMap
@mapping(id:"aws/mapping/brightnessSimpleRangeToBrightnessNumber"
  from:"aws/property/brightness/brightnessSimpleRange s",
  to:"aws/property/brightness/brightnessNumber t") {
  forward (
    expr: "s==Low => 20", # if boolean-expr true, map 20 to target t
    expr: "s==Medium => 45", # if boolean-expr true, map 45 to target t
    expr: "s==High => 75"     # if boolean-expr true, map 75 to target t
  )
  reverse (
    expr: "t.value >= 0 && t.value <= 33 => Low", #if expr-true, map Low to target s
    expr: "t.value > 33 && t.value <= 66 => Medium", # on true, map Medium to s
  )
}````
The following example creates a symmetrical mapping between to complex properties that represent color. It creates the following constructs:

- Two states that represent RGB and RGBW colors
- Two complex properties, one composed of the RGB state and another composed of the RGBW state
- A symmetrical mapping between the two representations

```plaintext
# Create state colorRGB
type colorRGBState @stateType(id:"urn:tdm:aws:state:colorRGB") {
  red   :int32 @property(id:"urn:tdm:aws:property:int32"), # int32 is the alias for the built-in property urn:tdm:aws:property:int32
  blue  :int32 @property(id:"urn:tdm:aws:property:int32"),
  green :int32 @property(id:"urn:tdm:aws:property:int32")
}

# Create state colorRGBW
type colorRGBWState @stateType(id:"urn:tdm:aws:state:colorRGBW") {
  red   :int32, @property(id:"urn:tdm:aws:property:int32")# int32 is alias for built-in property urn:tdm:aws:property:int32
  blue  :int32, @property(id:"urn:tdm:aws:property:int32")
  green :int32, @property(id:"urn:tdm:aws:property:int32")
  white :int32 @property(id:"urn:tdm:aws:property:int32")
}

# Create abstract property Color
type color @propertyType(id: "urn:tdm:aws:property:color") {ignore:void}

# Create complex property type from colorRGB state
type colorRGBProperty @propertyType(id:"urn:tdm:aws:property:color/colorRGB",
  dataType:"urn:tdm:aws:dataType:EntityType",
  instanceOf:"urn:tdm:aws:state:colorRGB") {ignore:void}

# Create complex property type from colorRGBW state
type colorRGBWProperty @propertyType(id:"urn:tdm:aws:property:color/colorRGBW",
  dataType:"urn:tdm:aws:dataType:EntityType",
  instanceOf:"urn:tdm:aws:state:colorRGBW") {ignore:void}

# Create mapping for complex properties colorRGB and colorRGBW
query colorRGBtoRGBWMapping @mapping(id:"urn:tdm:aws:mapping:colorRGBtoRGBW",
  From:"urn:tdm:aws:property:color/colorRGB f", To:"urn:tdm:aws:property:color/colorRGBW t")
{
  symmetric(
    map(expr:"f.red => t.red"), # For symmetric map, both expressions are property names.
    map(expr:"f.blue => t.blue"), # The s.blue property is mapped to the t.blue property.
    map(expr:"f.green => t.green") # The semTypeIds of left and right properties are either in the same
      property hierarchy or explicitly mapped.
  )
  forward # All properties used in Min() must be in same hierarchy or explicitly mapped
    map(expr:"Math.min(f.red, f.blue, f.green) => t.white")
}
```
The following example uses the mappings and properties from the previous two examples to create an additional complex mapping. It creates the following constructs.

- A state that contains the numerical range representation of brightness and the RGB representation of color
- A state that contains the enum representation of brightness and the RGBW representation of color
- An abstract property named brightColor
- A concrete complex property composed of the first state (numerical range brightness and RGB color)
- A concrete complex property composed of the second state (enum representation of brightness and RGBW color)
- A mapping between the two new complex properties

```graphql
# First state containing colorRGB and brightnessNumber properties
type brightColorState @stateType(id: "urn:tdm:aws:state:brightColor") {
  color: colorRGB @property(id: "urn:tdm:aws:property:color/colorRGB"),
  brightness: brightnessNumber @property(id:"urn:tdm:aws:property:brightness/ brightnessNumber")
}

# Second state containing colorRGBW and brightnessSimpleRange properties
type brightColorRGBWState @stateType(id: "urn:tdm:aws:state:brightColorRGBW") {
  colorRGBW:colorRGBW @property(id: "urn:tdm:aws:property:color/colorRGBW"),
  brightness:brightnessSimpleRange @property(id::"urn:tdm:aws:property:brightness/ brightnessSimpleRange")
}

# Abstract property brightColor
type color @propertyType(id: "urn:tdm:aws:property:brightColor") {ignore:void}

# Complex property brightColor1
type brightColor1Property @propertyType(id:"...:property:brightColor/brightColor1") {
  value:brightColorState
} {ignore:void}

# Complex property brightColorRGBW
type brightColorRGBWProperty @propertyType(id:"...:property:brightColor/brightColorRGBW") {
  value:brightColorRGBWState
} {ignore:void}

# Mapping between brightColor1Property <=> brightColorRGBWProperty
query brightColor1toRGBWMApping @mapping(id:"...:mapping:brightColor1toRGBW", from:"...:property:brightColor/brightColor1 f", to:"...:property:brightColor/brightColorRGBW t") {
  symmetric(
    expr:"f.color => t.colorRGBW", # re-uses mapping colorRGBtoRGBW
    expr:"f.brightness => t.brightness" # re-uses mapping
  )
}
```
Event

The Event construct describes a notification from a device that some action has been taken on it, such as a click of a camera. The structure of this construct contains the following information:

- The event name
- A unique identifier of the source device of the event
- A payload containing information that supports handling of the event

The following example creates an event that is published whenever a camera is clicked.

```typescript
type cameraClickedEvent @eventType(id: "urn:tdm:aws/sys/thing:Event:cameraClicked", payload: "urn:tdm:aws:Property:Boolean"){ignore:void}
```

Action

The Action construct is an abstract representation of a device performing an instance of its capability, such as a camera capturing an image. An Action takes properties as its parameters and returns properties as its output.

The following examples create three kinds of camera capture actions. The three actions vary according the types of parameters they take (if parameters are present). The first example describes the simplest kind of camera capture. The other two make the camera capture more complex by passing parameters that set a delay in milliseconds and picture resolution. All three return an image URL as output. The return keyword inside the braces and followed by a colon precedes the return value's name, type (@property directive), and URN.

```typescript
type cameraCaptureAction @actionType(id: "urn:tdm:aws:Action:cameraCapture"){
    return : imageLinkUri @property(id:"urn:tdm:aws:property:imageLink/imageLinkUri")
}
type cameraDelayedCaptureAction @actionType(id:"urn:tdm:aws:Action:cameraCapture/cameraDelayedCapture") {
    delay_ms : uint16 @property(id:"urn:tdm:aws:property:uint16"),
    return : imageLinkUri @property(id:"urn:tdm:aws:property:imageLink/imageLinkUri")
}
type cameraAdvancedCaptureAction @actionType(id:"urn:tdm:aws:Action:cameraCapture/cameraDelayedCapture/cameraAdvancedCapture") {
    delay_ms : uint16 @property(id:"urn:tdm:aws:property:uint16"),
    resolution : enumLowMediumHigh @property(id:"urn:tdm:aws:property:resolution/enumLowMediumHigh"),
    return : imageLinkUri @property(id:"urn:tdm:aws:property:imageLink/imageLinkUri")
}
```

Capability

The Capability construct describes a piece of functionality that is implemented by an IoT device. A Capability can extend one or more pre-existing Capabilities. It's a package containing a State and a set of Actions and Events. A Capability definition consists of the following:
Device model

The DeviceModel construct describes an abstraction of an IoT device or a stateful service. A DeviceModel must implement one Capability. It represents a conceptual device and isn't tied to any specific manufacturer.

A DeviceModel consists of the following:

- A TDM URN that identifies the device model
- A TDM URN that identifies the device model's Capability

The following example creates a camera device model that contains the camera capability created in the previous example.

```graphql
type cameraDevice @deviceModel(id:"urn:tdm:aws:deviceModel:camera", capability:"urn:tdm:aws:capability:camera") {ignore:void}
```

Device

The Device construct describes a specific IoT device that implements the Capability of a DeviceModel.

This is not a concrete device, but an abstract definition of a device. After a device is defined in TDM, concrete devices can be mapped to the device definition.

A Device definition includes the following:

- An implementation of the parent device model's State
- An implementation of the parent device model's Actions and Events in the context of a specific communication protocol, such as MQTT or Modbus
The following pseudocode describes what happens at a conceptual level when you create a device.

```
Create Device deviceUrn Device_Model deviceModelUrn '{
    [State '{' [alias propertyUrn]* '}' ]
    ([Action actionName Protocol [MQTT | ModBus] ] '{' action-protocol-body '}' )*)
    ([Event eventName Protocol [MQTT | ModBus] ] '{' event-protocol-body '}' )*)
'}

MQTT_action_protocol_body :: PROTOCOL MQTT2 '{
    REQUEST (TOPIC topicPath) Params '([' [paramName] paramPropertyUrn valueExpr ] [,])*' ')
    RESPONSE (TOPIC topicPath) Property returnPropertyUrn ')

MQTT_event_protocol_body :: PROTOCOL MQTT2 '{
    PAYLOAD [NAME payloadFieldName] TOPIC topicPath payloadPropertyUrn ')

Key Concepts

deviceUrn
   The URN that uniquely identifies the device.

deviceModelUrn
   The URN that uniquely identifies the device model from which the device inherits.

actionName
   The name of the action that the device implements.

eventName
   The name of the event that the device implements.

PROTOCOL
   The communication protocol that the device uses.

topicPath
   The name of the topic associated with a request, response, or event.

paramName
   The name of the action request parameter.

paramPropertyUrn
   The URN that specifies the parameter property.

valueExpr
   The expression that specifies the value to pass as a parameter value.

payloadFieldName
   Optional payload field name. Usually the payload is an unnamed structure.

payloadPropertyUrn
   The URN of the event payload property.

GraphQL Example
The following example creates a camera device that implements the camera device model.

```graphql
query cameraDevice @device(id: "urn:tdm:aws:device:camera",
  # Device URN
deviceModel: "urn:tdm:aws:deviceModel:cameraModel") {
  # Device model URN
  MQTT {
    # Communication protocol
    Capability(id: "urn:tdm:aws:capability:CameraCap") {
      # URN of capability.
      state {
        # State implementation
        lastClickedImage(name: "lastImage", property: "urn:tdm:aws:Property:Uri")
      }
      Action(name: "capture") {
        # Action name and implementation
        Publish {
          Request(topic: "camera/capture") {
            # MQTT request topic
            params
          }
          Response(topic: "camera/capture/finished") {
            # MQTT response topic
            responsePayload(property: "urn:tdm:aws:Property:Uri")
            # MQTT response payload (image URI)
          }
        }
      }
      Event(name: "clicked") {
        # Event implementation
        property: boolean @property(id: "urn:tdm:aws:Property:boolean")
      }
    }
  }
}
```

This example contains the following key elements.

- The URN that uniquely identifies the device (first argument).
- The URN that uniquely identifies the device model (second argument).
- The communication protocol used by the device. Possible values are ModBus (for local communication), HTTP (for remote communication), and MQTT (for both local and remote communication).
- The name and unique identifier (URN) of the capability that the device implements.
- The Action name and implementation.
- The Request definition. This definition specifies the MQTT request topic and the parameters, if any, that are sent to it.
- The Response definition. This definition specifies the MQTT response topic and the payload (an image URL) that is sent to it.
- The Event implementation.

See the DeviceActivity implementations in Workflow for sample implementations of Devices.

### Service

The Service construct describes either an AWS Lambda or a RESTful web service that can be called from a Workflow.
Conceptually a Service is analogous to a Device, because it can be called inside a workflow step. The structure of a Service is also similar to that of a Device. The primary difference is that the Action of a Service is a call to the web service or an invocation of a Lambda function. The Service structure contains the input and output parameters of the web service or Lambda call.

The following example creates a Service that invokes a Lambda function. The Lambda function uses Amazon Rekognition to match an image passed from the workflow with another image.

```graphql
query MlLambdas @service(id: "urn:tdm:aws:service:MlLambdas") {
  # Service URN
  AwsLambda {
    # Service type
    MlLambdasCap(id: "urn:tdm:aws:capability:MlLambdasCap") {
      # Name and URN of capability
      Action(name: "matchImage") {
        # Action name and implementation
        InvokeCloudLambda {
            # Lambda function ARN
            params {
              param(name: "imageUri", property: "urn:tdm:aws:property:Uri", value: "url")
            } // Request parameters
          } // Request
          Response {
            responsePayload(property: "urn:tdm:aws:property:Boolean")
            # Response payload type
          } // Response
        } // InvokeCloudLambda
      } // MlLambdasCap
    } // AwsLambda
  } // MlLambdas
}
```

This example contains the following key elements:

- The URN that uniquely identifies the service.
- The service type. Possible values are AwsLambda and REST.
- The name and unique identifier (URN) of the capability that the service implements.
- The Action name and implementation.
- The InvokeCloudLambda action specification. In an AWSLambda service, possible values are InvokeCloudLambda and InvokeGreengrassLambda.
- The Request definition, which specifies the payload that is sent to the Lambda function. This definition contains the following elements:
  - The Amazon Resource Name (ARN) that specifies the Lambda function to call.
  - The list of parameters to send as the request payload. This example sends an image URI to the Lambda function to determine whether the image is a match.
- The Response payload definition. This specifies the payload type whenever a response is expected.

The following example creates a Service that calls a web service. The web service creates a queue and then sends a message that contains the queue name.

```graphql
query Sqs @service(id: "urn:tdm:aws:service:Sqs") {
  # Service URN
  Sqs {
    # Service type
    SqsCap(id: "urn:tdm:aws:capability:SqsCap") {
      # Name and URN of capability
      Action(name: "createQueue") {
        # Action name and implementation
        CreateQueue {
          Request {
            queueName: "testQueue"
          } // Request
        } // CreateQueue
      } // SqsCap
    } // Sqs
  } // MlLambdas
}
```
REST {
  # Service type
  SqsCap(id: "urn:tdm:aws:capability:SqsCap") {
    # Name and URN of capability
    Action(name: "createQueue") {
      HttpGet {
        # Action specification (REST verb)
        Request(url: "https://sqs.us-west-2.amazonaws.com/", auth: "SigV4") {
          # REST endpoint. Authentication type
          (SigV4)
          headerParams {
            # Header parameters
            param(name: "host", property: "urn:tdm:aws:property:String", value: "sqs.us-west-2.amazonaws.com")
            param(name: "x-amz-date", property: "urn:tdm:aws:property:String", value: "#{tdm.lib.getDate()}")
          }
          queryParams {
            # Query parameters
            param(name: "QueueName", property: "urn:tdm:aws:property:String", value: "queueName")
          }
          Response {
            responsePayload(property: "urn:tdm:aws:property:Boolean")
            # Response payload type
          }
        }
      }
      } 
    }
    Action(name: "sendMessage") {
      HttpPost {
        # Action specification (REST verb)
          headerParams {
            # Header parameters
            param(name: "host", property: "urn:tdm:aws:property:String", value: "sqs.us-west-2.amazonaws.com")
            param(name: "x-amz-date", property: "urn:tdm:aws:property:String", value: "#{tdm.lib.getDate()}")
          }
          bodyParams {
            # Body parameters
            param(name: "title", property: "urn:tdm:aws:property:String", value: "title")
            param(name: "body", property: "urn:tdm:aws:property:String", value: "queueName")
          }
          Response {
            responsePayload(property: "urn:tdm:aws:property:Boolean")
            # Response payload type
          }
        }
      }
    }
  }
}
This example contains the following key elements that differ from the AWS Lambda service.

- A REST service type.
- The action specifications. These map to the corresponding REST verbs. Possible values are `HttpGet` and `HttpPost`.
- The `Request` definition contains two new arguments:
  - The URL to which the REST request will be sent.
  - The type of authentication to use when making the REST request.
- The `Request` definition contains three kinds of parameter lists:
  - A list of `headerParams` that defines the request headers.
  - A list of `queryParams` that defines the query string parameters.
  - A list of `bodyParams` that defines the request body.

  **Note**
  Some of these parameters use path expressions to get the current date and time, as well as macro expressions to interpolate the resulting values into strings. For more information, see Expressions.

You implement a service in a Workflow in a way that is nearly identical to a device implementation. See the `WebServiceActivity` implementation in Workflow for a sample implementation of a Service.

## Workflow

The Workflow construct (also called flow) describes an automated process that consists of multiple devices and stateful services.

Workflow takes a set of parameters and consists of an array of steps that are connected to events. Input events can trigger a step, and a step can generate an output event. Each step can represent a Lambda function, a device action, or a web application method.

The following pseudocode defines what happens at a conceptual level when you create a workflow.

```python
CREATE Workflow workflowUrn '{' [paramName paramType [= defaultValue] [,]]* '}' { [varName varType [= defaultValue] [,]]* STEPS { [STEP (NAME stepName, [CONDITION conditionExpr] [DESCRIPTION descriptionText]) { [EVENTS { [INPUT ( ${expression-over events} ) ] [OUTPUT ( [eventName, [eventUrn] [,]]* )] } ] [ACTIVITY ('' [paramName = paramExpr [,]]* [,] out =$varName '') [OnError {Retries(Retry (name=retryName, condition=booleanExpr, [ [interval=expr] [backoffRate=expr] [maxAttempts=expr] )} * Catches(Catch(name=catchName, condition=booleanExpr, raiseEvent([eventName [,]]* ))) ] } [OnSuccess { (CHECK (name=chkName, condition=booleanExpr) THEN (onPassExpr, onFailExpr) * ]
```
Key concepts
workflowUrn
    Unique identifier of the workflow.
paramType
    Data type of the workflow parameter.
paramName
    Workflow parameter name.
defaultValue
    Default parameter value.
varName
    Variable name.
varType
    Variable type.
stepName
    Name of a step in the workflow. Unique to the workflow.
descriptionText
    Optional description.
conditionExpr
    Optional condition. If it resolves to false, the step is skipped.
eventName
    Input or output event name. The step either waits for it to start, or publishes it on completion.
eventUrn
    Optional global event URN.
ACTIVITY
    Placeholder for device activity, web activity, or a built-in control activity.
booleanExpr
    A Boolean condition.
onPassExpr
    Evaluated if CHECK(booleanExpr) evaluates to true.
onFailExpr
    Evaluated if CHECK(booleanExpr) evaluates to false.
onError, retries, retry, catches, catch
    Error handling with the same semantics as AWS Step Functions.
    OnError is evaluated when the activity fails or times out. The Retries are evaluated in the specified order. On exhaustion of all Retries, control falls to the Catch clause.
GraphQL Example

The following GraphQL creates a workflow that monitors door entry in a building. A camera takes a picture of the person entering the building, which triggers a face detection service. The face detection service then displays an image of the person on a screen. The example also handles camera errors and failure.

```graphql
query monitorDoorEntry ($cameraId: DeviceId!, $screenId: DeviceId!, $lampId: DeviceId!, $cameraDelay_ms : Number = 0, $cameraFailedImage: String = "http://abc/defaultCameraFailedImage.jpg")
{ # @workflowType directive and workflow unique URN

  # Declare workflow variables
  variables {
    # Workflow variables
    cameraImage  @property(id:"urn:tdm:aws:property:image/imageUri")
    # Semantic property imageUri
    faceDetected @property(id:"urn:tdm:aws:property:bool/boolean")
    # Raw datatype property boolean
  }

  steps {
    # Set of steps.
    # Device activity step
    step(name:"camera", description:"Activate Camera" outEvent: ["CameraClicked"]){
      # Step name, description, and output event
      DeviceActivity(deviceModel:"urn:tdm:aws:device:camera" deviceId:"$cameraId"){
        # Device activity ARN. Device ID
        capture(delay_ms:$cameraDelay_ms)
      } OnError {
        # Error handling
        Retries {
          Retry(name:"foo1", errorEquals:["ErrorA", "ErrorB"], interval:"1 sec", backoffRate:2.0, maxAttempts:2),
          Retry(name:"foo2", errorEquals:["ErrorC"], interval:"5 sec"),
          Retry(name:"foo3", condition:"${Error IN ('ErrorD', 'ErrorE')}", interval:"5 sec")
        } Catches {
          Catch(name:'catch1',condition:"${Error=='ErrorE'}", expr:"$cameraFailed.IsTrue=true"]
        }
      }
      OnSuccess {
        # Validation of image captured by the camera
        Check(name:"validation1", condition:"${lib.fileSize(cameraImage) > 100}" ) {
          OnPass { stmt(expr:"${imageOK=true}") }
          OnFail { stmt(expr:"$cameraFailed.IsTrue=true")
            # Raise cameraFailed Event
          }
        }
      }
    }
    # Device activity step
    step (name:"Lamp") {
      # Step name
      DeviceActivity(deviceModel:"urn:tdm:aws:device:lightBulb",deviceId:"$lampId"){
        # Device activity ARN. Device ID
        powerOn(param:$cameraDelay_ms)
      }
    }
  }
}
```
This example contains the following key elements that differ from the AWS Lambda service:

- A set of arguments and their data types. This example passes three non-nullable device URNs, a camera burst delay setting with a default value of 0, and an image URL (with a default value) for failed image captures.
- The workflowType directive and its unique URN.
- A set of workflow variables and their data types (and property URNs).
- A list of steps with names and descriptions.
- Each step has an activity specification. Possible values are DeviceActivity and WebServiceActivity.
- A DeviceActivity is specified by a device model and the ID of the concrete device in the workflow.
- The device or service activity that the device or service in each step performs. These activities are specified in the device model's capability. They are implemented in the concrete device's definition.
- Error handling with the OnError, Retries, Retry, Catches, and Catch. The OnError clause is evaluated when the activity fails or times out. The Retries clause is then evaluated in the specified order until all Retry clauses have been attempted. If all of the retries fail, control falls to the Catches clause. These clauses have the same semantics as Retry and Catch in AWS Step Functions. The OnSuccess clause also contains additional validation to ensure that the camera captured a valid image.

The workflow makes extensive use of Expressions to determine how the workflow proceeds, depending on whether the camera works and the condition of the image that it returns. The web service activity, for example, doesn't run unless a valid image is captured. See Expressions for more information about how to use expressions.

System

The System construct describes a collection of devices, services, and workflows that interact with each other in an IoT system.
For example, a security system can consist of entry sensors, cameras, light bulbs, and a door monitoring workflow. A System can be composed of other systems to create arbitrarily complex systems of systems.

The following pseudocode shows what happens at a conceptual level when you create a system.

```
CREATE SYSTEM systemUrn [Description text] '{' [deviceRole deviceUrn [Description text] [,]]* [serviceRole serviceUrn [Description text] [,]]* [workflowRole flowUrn [Description text] [,]]* '}
```

**Key concepts**

- **systemUrn**
  - Unique URN of the system.
- **deviceRole deviceUrn**
  - Role or name of the device identified by the deviceUrn.
- **serviceRole serviceUrn**
  - Role or name of the service identified by serviceUrn.
- **workflowRole flowUrn**
  - Role or name of the workflow identified by the workflowUrn.

**GraphQL Example**

The following GraphQL creates a door entry security system.

```graphql
type DoorEntrySecuritySystem @SystemType(id:"urn:tdm:aws:system:DoorEntrySecurity", description:"Door Entry Security")
{
  # Devices and Services: Begin
  entrySensor: MotionSensorDevice @thing(id: "urn:tdm:aws:device:motionSensor"),
  entryCamera: CameraDevice @thing(id: "urn:tdm:aws:device:camera"),
  cameraLight: LightBulbDevice @thing(id: "urn:tdm:aws:device:lightBulb"),
  faceDetection: FaceDetectionService @thing(id: "urn:tdm:aws:service:faceDetection"),
  # Devices and Services: End
  # PowerOnOff is a Switch that controls power to entrySensor – motionSensor device
  powerOnOff: SwitchOnOffDevice @thing(id: "urn:tdm:aws:device:switchOnOff"),
  # Internal flow that automatically kicks in when entrySensor is powered on
  doorSecurityFlow: MonitorDoorEntryFlow @workflow(id: "urn:tdm:aws:workflow:monitorDoorEntry"),
  # Export PowerOnOffFlow as DoorEntrySecurity public API
  powerOnOffFlow: PowerOnOffFlow @workflow(id: "urn:tdm:aws:workflow:monitorDoorEntry, params[{$powerOnOff, $entrySensor}]"),
  # Expr property (implicitly isReadOnly=true)
  IsPowerOn : Boolean! @expr("powerOnOff.state.power && entrySensor.state.connected")
}
```
The following example creates a system that represents a professional camera room. It contains a set of cameras and light bulbs, and a service that combines the images taken by the cameras into 3D visualizations.

```typescript
type CamaraRoom @systemType(id: "urn:tdm:aws:system:camera/3DCameraRoom", description: "3DCameraRoom") {
  camera1: CameraDevice @Thing(id: "urn:tdm:aws:device:camera/advancedCamera"),
  cameraLight1: LightBulbDevice @thing(id:"urn:tdm:aws:device:lightbulb/cameraBulb"),
  camera2: CameraDevice @Thing(id: "urn:tdm:aws:device:camera/advancedCamera"),
  cameraLight2: LightBulbDevice @thing(id: "urn:tdm:aws:device:lightbulb/cameraBulb"),
  camera3: CameraDevice @Thing(id: "urn:tdm:aws:device:camera/advancedCamera"),
  cameraLight3: LightBulbDevice @thing(id: "urn:tdm:aws:device:lightbulb/cameraBulb"),
  camera4: CameraDevice @Thing(id: "urn:tdm:aws:device:camera/advancedCamera"),
  cameraLight4: LightBulbDevice @thing(id: "urn:tdm:aws:device:lightbulb/cameraBulb"),
  combineImages:CombineImagesFor3DVisualizationService
    @service(id:"urn:tdm:aws:service:combineImagesFor3DVisualization")
}
```

### Trigger

The Trigger construct defines the conditions that start a workflow.

Workflow can take parameters, including the device IDs to use in the workflow and other variables that help dictate the flow of control. The parameters of a workflow are specified in the System deployment configuration (SDC). SDC triggers have two components: a condition and some number of actions. The condition specifies whether to trigger a new flow, and an action specifies exactly what to do if the condition is true.

Because the parameters and triggers for a workflow are defined in a Deployment configuration, expressions are also useful in that context. Triggers have two components: a condition and a set of one or more actions. The condition specifies whether to trigger a new workflow, and an action specifies what the workflow does if the condition is true. The following example shows some ways of using expressions in the context of deployment configurations. This example also appears in Deployment configuration.

```typescript
  entrySensor(deviceId: "ABC123")
  entryCamera(deviceId: "XYZ987")
  triggers {
    MotionDetectedTrigger(description: "an example trigger") {
      condition(expr: "devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value")          # Path expressions used inside a predicate expression
      action(expr: "ThingsGraph.startFlow('motionDetectionFlow', devices.entrySensor.motionEvent")") # Initiate workflow
    }
    MotionEndedTrigger(description: "an example trigger") {
      condition(expr: "devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value == false") # Path expressions used inside a predicate expression.
```
The first condition uses two path expressions to identify a device that is used in a workflow and one of that device's events, with a predicate expression that determines whether the device has detected motion. The `devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value` expression evaluates to true if the `lastEvent` value sent by the `entrySensor` device is true, a non-empty string, or a numeric value other than 0. This signifies that a motion detected event has occurred.

The second condition uses the same combination of path expressions inside a predicate expression to determine whether a given motion has ended. The `devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value == false` evaluates to true if the `lastEvent` value sent by the `entrySensor` device is false, an empty string, or 0. This signifies that the sensor is no longer detecting motion.

The actions specified in this definition use the `ThingsGraph.startFlow` function, which initiates the specified workflow and, where necessary, sends the `devices.entrySensor.motionEvent` to the workflow.

TDM also supports time intervals as conditional expressions in triggers. For example, if you want an action to occur every 5 minutes, you can specify that interval in the condition, as in the following example.

```
condition(expr: "every 5 minutes")
```

You can express the duration of a time interval in a condition expression with the following values:

- Seconds
- Minutes
- Hours
- Days
- Weeks
- Months
- Years

The actions specified in this definition use the `ThingsGraph.startFlow` function, which initiates the specified workflow and, where necessary, sends the `devices.entrySensor.motionEvent` to the workflow.

For more information about the types of expressions that you can use in TDM, see Expressions.

## Deployments

The `System` construct creates an abstract structure that consists of devices and services.
The devices implemented in a specific instance of a System have a physical presence. TDM supplies constructs that associate Systems with physical structures and that arrange the positioning of devices in those structures.

TDM includes modeling for the following concepts:

- System deployments for specific physical structures

The topics in this section describe the constructs that TDM provides for locating systems in physical locations.

**Deployment**

The Deployment construct associates a physical location with specific devices and the triggers that start the workflows in which they are used.

The following example creates a Deployment for the door entry security system.

**Note**
Note that this example makes extensive use of Expressions. For more information about how to construct and use expressions, see Expressions.

```graphql
    entrySensor(deviceId: "ABC123")
    entryCamera(deviceId: "XYZ987")

    triggers {
        MotionDetectedTrigger(description: "an example trigger") {
            condition(expr: "devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value") # Path expressions used inside a predicate expression
            action(expr: "ThingsGraph.startFlow('motionDetectionFlow', devices.entrySensor.motionEvent") # Initiate workflow
        }
        MotionEndedTrigger(description: "an example trigger") {
            condition(expr: "devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value == false") # Path expressions used inside a predicate expression
            action(expr: "ThingsGraph.startFlow('motionDetectionFlow', devices.entrySensor.motionEvent") # Initiate workflow
            action(expr: "ThingsGraph.startFlow('logEventFlow')") # Initiate workflow
        }
    }
}
```

The first condition uses two path expressions to identify a device that is used in a workflow and one of that device's events, with a predicate expression that determines whether the device has detected motion. The `devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value` expression evaluates to true if the last event value sent by the entrySensor device is true, a non-empty string, or a numeric value other than 0. This signifies that a motion detected event has occurred.

The second condition uses the same combination of path expressions inside a predicate expression to determine whether a given motion has ended. The `devices[name == 'entrySensor'].events[name == 'motionDetected'].lastEvent.value == false`
evaluates to true if the `lastEvent` value sent by the `entrySensor` device is false, an empty string, or 0. This signifies that the sensor is no longer detecting motion.

TDM also supports time intervals as conditional expressions. For example, if you want an action to occur every 5 minutes, you can specify that interval in the condition, as in the following example.

```
condition(expr: "every 5 minutes")
```

You can express the duration of a time interval in a condition expression with the following values:

- Seconds
- Minutes
- Hours
- Days
- Weeks
- Months
- Years

The actions specified in this definition use the `ThingsGraph.startFlow` function. This function initiates the specified workflow and, where necessary, sends the `devices.entrySensor.motionEvent` to the workflow.
Tagging your AWS IoT Things Graph resources

To help you manage and organize your system instances, you can optionally assign your own metadata to each of these resources in the form of tags.

This topic describes tags and shows you how to create them.

Tag basics

Tags enable you to categorize your AWS IoT Things Graph 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 specific 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 could define a set of tags for your system instances that helps you track them by location. 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.

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 by using the AWS CLI and the AWS IoT Things Graph API. You can associate tags with system instances when you create them by using the "tags" field in the following command:

• CreateSystemInstance

You can add, modify, or delete tags for existing resources that support tagging by using the following commands:

• TagResource
• ListTagsForResource
• UntagResource

You should know the following.

• 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.

For more information, see AWS tagging strategies.

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.
• Don’t use the "aws:" prefix in your tag names or values because 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. Generally, allowed characters are letters (uppercase and lowercase), spaces, and numbers that can be represented in UTF-8, and the following special characters: + - = . _ : /

Using tags with IAM policies

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:

• aws:ResourceTag/tag-key: tag-value – Allow or deny user actions on resources with specific tags.
• aws:RequestTag/tag-key: tag-value – Require that a specific tag be used (or not used) when making an API request to create or modify a resource that allows tags.
• aws:TagKeys: [tag-key, ...] – 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 that’s capable of being tagged is a required parameter. For example, the use of CreateFlowTemplate will not be 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.

Controlling access using tags in the AWS Identity and Access Management User Guide has additional information about using tags. 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. An IAM user restricted by this policy cannot do the following:

• Give a resource the tag "env=prod" (in the example, see the line "aws:RequestTag/env" : "prod")
• Modify or access a resource that has an existing tag "env=prod" (in the example, see the line "aws:ResourceTag/env" : "prod")

```json
{
  "Version" : "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Deny",
```

254
"Action": "iotthingsgraph:*",
"Resource": "*",
"Condition": {
  "StringEquals": {
    "aws:RequestTag/env": "prod"
  }
},
{
  "Effect": "Deny",
  "Action": "iotthingsgraph:*",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "aws:ResourceTag/env": "prod"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "iotthingsgraph:*"
  ],
  "Resource": "*"
}
}

You can also specify multiple tag values for a given tag key by enclosing them in a list, as shown.

"StringEquals": {
  "aws:ResourceTag/env": ["dev", "test"]
}

**Note**

If you use tags to allow or deny users access to resources, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, a user could circumvent your restrictions and gain access to a resource by modifying its tags.
AWS CloudFormation support for AWS IoT Things Graph

AWS CloudFormation is a service that can help you create, manage, and replicate your AWS resources. You can use AWS CloudFormation templates to define AWS IoT Things Graph flows (workflows) that you want to deploy.

The resources and infrastructure that you generate from a template is called a stack. You can define all of your resources in one template or refer to resources from other stacks. For more information about AWS CloudFormation templates and features, see What is AWS CloudFormation? in the AWS CloudFormation User Guide.

Creating resources

AWS CloudFormation templates are JSON or YAML documents that describe the properties and relationships of AWS resources.

The following AWS IoT Things Graph resources are supported:

- Flows (workflows)

In AWS CloudFormation templates, the structure and syntax of AWS IoT Things Graph resources are based on the AWS IoT Things Graph API. For example, you create a flow by assigning values to the definition and compatibleNamespaceVersion parameters. For more information, see AWS IoT Things Graph API Reference.
Glossary

Action

An abstract representation of a device that performs an instance of its capability, such as a camera capturing an image.

*Camera example:* The Camera device model in the AWS IoT Things Graph console has a capture action.

Capability

An abstract representation of a specific piece of functionality that a device can perform, such as the ability to capture an image.

*Motion sensor example:* The motion sensor device model in the AWS IoT Things Graph console has a MotionSensorCapability that contains its capture action.

Device model

An abstract representation of a device as a set of actions, events, and states. A model separates a device interface from its underlying implementation. Models are like building blocks that you can snap together and integrate into any number of IoT applications. IoT applications can interact with the model (and thereby the underlying device) by using the device capabilities that it exposes. AWS IoT Things Graph includes built-in models. Users can also create their own with the AWS IoT Things Graph model editor. Devices inherit from the device model and implement its capability.

*Motion sensor example:* You can find the motion sensor device model in the AWS IoT Things Graph console. The Aukru HCSR501 motion sensor device (also available in the AWS IoT Things Graph console) implements the generic motion sensor device model.

Device

An implementation (commonly from a manufacturer) of a device model. It inherits from a device model and implements the capability associated with the device model. A specific camera produced by one manufacturer inherits from a generic camera device model.

You associate things in your registry by using devices.

*Motion sensor example:* The Aukru HCSR501 motion sensor is an example of a device. You can find it in the AWS IoT Things Graph console. This device inherits from the motion sensor device model that is also available in the AWS IoT Things Graph console.

Event

An abstract representation of a change in some condition or of an action that was taken on it, such as the click of a camera.

*Motion sensor example:* The motion sensor device model in the AWS IoT Things Graph console has aStateChanged event that occurs whenever the sensor detects motion.

Flow (or workflow)

A definition of the logical interactions and orders of execution between the devices and services. Flows consist of device models and services. Flows define how the devices and services interact after a triggering event occurs. A flow lists these interactions as a sequence of steps. Each step contains an action on a device or web service, and the related inputs to and outputs from that action.

For more information, see *How a flow works.*
Flow configuration

An implementation a workflow (flow) for a specific location or deployment. A flow configuration consists of a flow, associated triggers, and all the corresponding physical things that interact with each other in the flows.

For more information, see How a flow configuration works.

Mapping

An information model that enables AWS IoT Things Graph to convert the output message of one device into the expected format for the next device in the flow. Mappings bridge differences between the devices and enable them to work together. Mappings enable you to build IoT applications that use a variety of devices from different manufacturers.

AWS IoT Things Graph includes mappings for basic types, such as Float32ToFloat64 and JSONToString.

Namespace

The container for entities (for example, models), created using the Things Graph Data Modeling (TDM) language, that are associated with a specific account. The namespace is part of the URN for each entity. When you upload TDM entities for the first time, AWS IoT Things Graph creates a namespace for you. The namespace consists of your AWS Region, your account ID, and default. The pattern is REGION/ACCOUNT_ID/default. For example, a TDM instance created in the us-west-2 Region looks like this: us-west-2/012345678910/default.

Each user has a unique private namespace that you can synchronize with the public namespace.

See also AWS IoT Things Graph namespaces.

Node

A visual representation of a workflow step in the AWS IoT Things Graph workflow designer.

Property

The basic building block of a state, which describes the state of a device, such as its color or accuracy.

AWS IoT Things Graph provides several built-in simple properties, such as Int32.

Service model

A web service or an AWS Lambda function that you can call from a workflow. Its role in a workflow is analogous to that of a device (p. 257) or device model (p. 257).

The Rekognition service model in the AWS IoT Things Graph console exposes the Amazon Rekognition DetectFaces API.

State

A set of properties that describe a device or a web service.

The motion sensor device model in the AWS IoT Things Graph console has a MotionSensorState. This contains a Boolean property that indicates whether the sensor has detected motion.

TDM URN

A URN that provides a hierarchical way of typing devices, properties, and entities. It contains organizational information (such as physical location) about a type, and semantic information (such as color or the URI for an image).

The following TDM URN is the ID of the Aukru HCSR501 motion sensor device.

urn:tdm:aws/examples:device:HCSR501MotionSensor
For more information, see TDM URN scheme.

Workflow

See Flow (or workflow (p. 257))