Developer Guide

Amazon API Gateway

Amazon API Gateway: Developer Guide

Table of Contents

What is Amazon API Gateway?

> **Note**
>
> The redesigned API Gateway console experience is now available. For a tutorial on how to use the console to create a REST API, see Getting started with the REST API console.

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. As an API Gateway API developer, you can create APIs for use in your own client applications. Or you can make your APIs available to third-party app developers. For more information, see the section called “Who uses API Gateway?”.

API Gateway creates RESTful APIs that:

- Are HTTP-based.
- Enable stateless client-server communication.
- Implement standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE.

For more information about API Gateway REST APIs and HTTP APIs, see the section called “Choosing between REST APIs and HTTP APIs ”, *Working with HTTP APIs*, the section called “Use API Gateway to create REST APIs”, and the section called “Create and configure”.

API Gateway creates WebSocket APIs that:

- Adhere to the WebSocket protocol, which enables stateful, full-duplex communication between client and server.
- Route incoming messages based on message content.

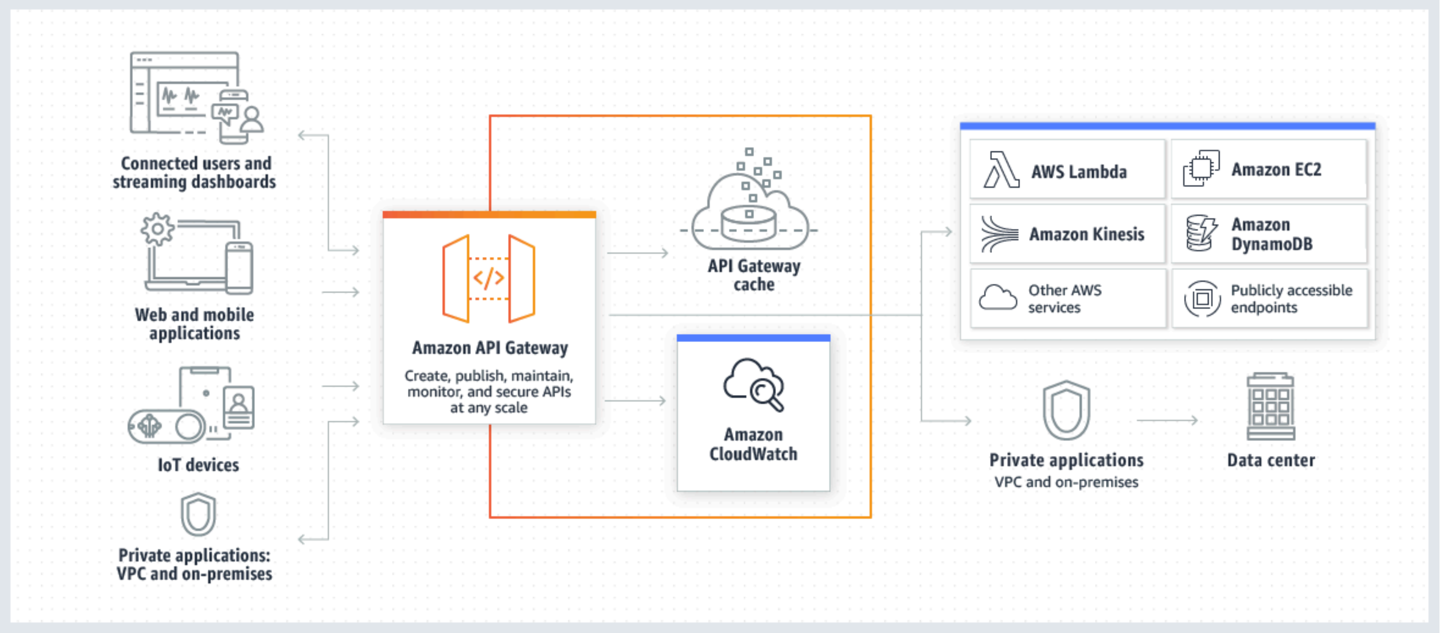
For more information about API Gateway WebSocket APIs, see the section called “Use API Gateway to create WebSocket APIs” and the section called “About WebSocket APIs”.

Topics

- Features of API Gateway
- API Gateway use cases
- Accessing API Gateway
- Part of AWS serverless infrastructure
- How to get started with Amazon API Gateway
- Amazon API Gateway concepts
- Choosing between REST APIs and HTTP APIs
- Getting started with the REST API console

Architecture of API Gateway

The following diagram shows API Gateway architecture.


Connected users and streaming dashboards
Web and mobile applications
IoT devices
Private applications: VPC and on-premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private applications
VPC and on-premises
Data center

This diagram illustrates how the APIs you build in Amazon API Gateway provide you or your developer customers with an integrated and consistent developer experience for building AWS serverless applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls. These tasks include traffic management, authorization and access control, monitoring, and API version management.

API Gateway acts as a "front door" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon Elastic Compute Cloud

(Amazon EC2), code running on AWS Lambda, any web application, or real-time communication applications.

Features of API Gateway

Amazon API Gateway offers features such as the following:

- Support for stateful (WebSocket) and stateless (HTTP and REST) APIs.
- Powerful, flexible authentication mechanisms, such as AWS Identity and Access Management policies, Lambda authorizer functions, and Amazon Cognito user pools.
- Canary release deployments for safely rolling out changes.
- CloudTrail logging and monitoring of API usage and API changes.
- CloudWatch access logging and execution logging, including the ability to set alarms. For more information, see the section called “CloudWatch metrics” and the section called “Metrics”.
- Ability to use AWS CloudFormation templates to enable API creation. For more information, see Amazon API Gateway Resource Types Reference and Amazon API Gateway V2 Resource Types Reference.
- Support for custom domain names.
- Integration with AWS WAF for protecting your APIs against common web exploits.
- Integration with AWS X-Ray for understanding and triaging performance latencies.

For a complete list of API Gateway feature releases, see *Document history*.

API Gateway use cases

Topics

Use API Gateway to create REST APIs

An API Gateway REST API is made up of resources and methods. A resource is a logical entity that an app can access through a resource path. A method corresponds to a REST API request that is submitted by the user of your API and the response returned to the user.

For example, `/incomes` could be the path of a resource representing the income of the app user. A resource can have one or more operations that are defined by appropriate HTTP verbs such as GET, POST, PUT, PATCH, and DELETE. A combination of a resource path and an operation identifies a method of the API. For example, a `POST /incomes` method could add an income earned by the caller, and a `GET /expenses` method could query the reported expenses incurred by the caller.

The app doesn't need to know where the requested data is stored and fetched from on the backend. In API Gateway REST APIs, the frontend is encapsulated by *method requests* and *method responses*. The API interfaces with the backend by means of *integration requests* and *integration responses*.

For example, with DynamoDB as the backend, the API developer sets up the integration request to forward the incoming method request to the chosen backend. The setup includes specifications of an appropriate DynamoDB action, required IAM role and policies, and required input data transformation. The backend returns the result to API Gateway as an integration response.

To route the integration response to an appropriate method response (of a given HTTP status code) to the client, you can configure the integration response to map required response parameters from integration to method. You then translate the output data format of the backend to that of the frontend, if necessary. API Gateway enables you to define a schema or model for the payload to facilitate setting up the body mapping template.

API Gateway provides REST API management functionality such as the following:

- Support for generating SDKs and creating API documentation using API Gateway extensions to OpenAPI
- Throttling of HTTP requests

Use API Gateway to create HTTP APIs

HTTP APIs enable you to create RESTful APIs with lower latency and lower cost than REST APIs.

You can use HTTP APIs to send requests to AWS Lambda functions or to any publicly routable HTTP endpoint.

For example, you can create an HTTP API that integrates with a Lambda function on the backend. When a client calls your API, API Gateway sends the request to the Lambda function and returns the function's response to the client.

HTTP APIs support OpenID Connect and OAuth 2.0 authorization. They come with built-in support for cross-origin resource sharing (CORS) and automatic deployments.

To learn more, see the section called “Choosing between REST APIs and HTTP APIs ”.

Use API Gateway to create WebSocket APIs

In a WebSocket API, the client and the server can both send messages to each other at any time. Backend servers can easily push data to connected users and devices, avoiding the need to implement complex polling mechanisms.

For example, you could build a serverless application using an API Gateway WebSocket API and AWS Lambda to send and receive messages to and from individual users or groups of users in a chat room. Or you could invoke backend services such as AWS Lambda, Amazon Kinesis, or an HTTP endpoint based on message content.

You can use API Gateway WebSocket APIs to build secure, real-time communication applications without having to provision or manage any servers to manage connections or large-scale data exchanges. Targeted use cases include real-time applications such as the following:

- Chat applications
- Real-time dashboards such as stock tickers
- Real-time alerts and notifications

API Gateway provides WebSocket API management functionality such as the following:

- Monitoring and throttling of connections and messages
- Using AWS X-Ray to trace messages as they travel through the APIs to backend services
- Easy integration with HTTP/HTTPS endpoints

Who uses API Gateway?

There are two kinds of developers who use API Gateway: API developers and app developers.

An API developer creates and deploys an API to enable the required functionality in API Gateway. The API developer must be a user in the AWS account that owns the API.

An app developer builds a functioning application to call AWS services by invoking a WebSocket or REST API created by an API developer in API Gateway.

The app developer is the customer of the API developer. The app developer doesn't need to have an AWS account, provided that the API either doesn't require IAM permissions or supports authorization of users through third-party federated identity providers supported by Amazon Cognito user pool identity federation. Such identity providers include Amazon, Amazon Cognito user pools, Facebook, and Google.

Creating and managing an API Gateway API

An API developer works with the API Gateway service component for API management, named `apigateway`, to create, configure, and deploy an API.

As an API developer, you can create and manage an API by using the API Gateway console, described in Getting started with API Gateway, or by calling the API references. There are several ways to call this API. They include using the AWS Command Line Interface (AWS CLI), or by using an AWS SDK. In addition, you can enable API creation with AWS CloudFormation templates or (in the case of REST APIs and HTTP APIs) Working with API Gateway extensions to OpenAPI.

For a list of Regions where API Gateway is available, as well as the associated control service endpoints, see Amazon API Gateway Endpoints and Quotas.

Calling an API Gateway API

An app developer works with the API Gateway service component for API execution, named `execute-api`, to invoke an API that was created or deployed in API Gateway. The underlying programming entities are exposed by the created API. There are several ways to call such an API. To learn more, see Invoking a REST API in Amazon API Gateway and Invoking a WebSocket API.

Accessing API Gateway

You can access Amazon API Gateway in the following ways:

- **AWS Management Console** – The AWS Management Console provides a web interface for creating and managing APIs. After you complete the steps in *Prerequisites*, you can access the API Gateway console at https://console.aws.amazon.com/apigateway.
- **AWS SDKs** – If you're using a programming language that AWS provides an SDK for, you can use an SDK to access API Gateway. SDKs simplify authentication, integrate easily with your development environment, and provide access to API Gateway commands. For more information, see Tools for Amazon Web Services.
- **API Gateway V1 and V2 APIs** – If you're using a programming language that an SDK isn't available for, see the Amazon API Gateway Version 1 API Reference and Amazon API Gateway Version 2 API Reference.
- **AWS Command Line Interface** – For more information, see Getting Set Up with the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.
- **AWS Tools for Windows PowerShell** – For more information, see Setting Up the AWS Tools for Windows PowerShell in the *AWS Tools for Windows PowerShell User Guide*.

Part of AWS serverless infrastructure

Together with AWS Lambda, API Gateway forms the app-facing part of the AWS serverless infrastructure. To learn more about getting started with serverless, see the Serverless Developer Guide.

For an app to call publicly available AWS services, you can use Lambda to interact with required services and expose Lambda functions through API methods in API Gateway. AWS Lambda runs your code on a highly available computing infrastructure. It performs the necessary execution and administration of computing resources. To enable serverless applications, API Gateway supports streamlined proxy integrations with AWS Lambda and HTTP endpoints.

How to get started with Amazon API Gateway

For an introduction to Amazon API Gateway, see the following:

- *Getting started*, which provides a walkthrough for creating an HTTP API.
- Serverless land, which provides instructional videos.
- Happy Little API Shorts, which is a series of brief instructional videos.

Amazon API Gateway concepts

API Gateway

API Gateway is an AWS service that supports the following:

- Creating, deploying, and managing a RESTful application programming interface (API) to expose backend HTTP endpoints, AWS Lambda functions, or other AWS services.
- Creating, deploying, and managing a WebSocket API to expose AWS Lambda functions or other AWS services.
- Invoking exposed API methods through the frontend HTTP and WebSocket endpoints.

API Gateway REST API

A collection of HTTP resources and methods that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. You can deploy this collection in one or more stages. Typically, API resources are organized in a resource tree according to the application logic. Each API resource can expose one or more API methods that have unique HTTP verbs supported by API Gateway. For more information, see the section called “Choosing between REST APIs and HTTP APIs ”.

API Gateway HTTP API

A collection of routes and methods that are integrated with backend HTTP endpoints or Lambda functions. You can deploy this collection in one or more stages. Each route can expose one or more API methods that have unique HTTP verbs supported by API Gateway. For more information, see the section called “Choosing between REST APIs and HTTP APIs ”.

API Gateway WebSocket API

A collection of WebSocket routes and route keys that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. You can deploy this collection in one or more stages. API methods are invoked through frontend WebSocket connections that you can associate with a registered custom domain name.

API deployment

A point-in-time snapshot of your API Gateway API. To be available for clients to use, the deployment must be associated with one or more API stages.

API developer

Your AWS account that owns an API Gateway deployment (for example, a service provider that also supports programmatic access).

API endpoint

A hostname for an API in API Gateway that is deployed to a specific Region. The hostname is of the form `{api-id}.execute-api.{region}.amazonaws.com`. The following types of API endpoints are supported:

- Edge-optimized API endpoint
- Private API endpoint
- Regional API endpoint

API key

An alphanumeric string that API Gateway uses to identify an app developer who uses your REST or WebSocket API. API Gateway can generate API keys on your behalf, or you can import them from a CSV file. You can use API keys together with Lambda authorizers or usage plans to control access to your APIs.

See API endpoints.

API owner

See API developer.

API stage

A logical reference to a lifecycle state of your API (for example, 'dev', 'prod', 'beta', 'v2'). API stages are identified by API ID and stage name.

App developer

An app creator who may or may not have an AWS account and interacts with the API that you, the API developer, have deployed. App developers are your customers. An app developer is typically identified by an API key.

Callback URL

When a new client is connected to through a WebSocket connection, you can call an integration in API Gateway to store the client's callback URL. You can then use that callback URL to send messages to the client from the backend system.

Developer portal

An application that allows your customers to register, discover, and subscribe to your API products (API Gateway usage plans), manage their API keys, and view their usage metrics for your APIs.

Edge-optimized API endpoint

The default hostname of an API Gateway API that is deployed to the specified Region while using a CloudFront distribution to facilitate client access typically from across AWS Regions. API requests are routed to the nearest CloudFront Point of Presence (POP), which typically improves connection time for geographically diverse clients.

See API endpoints.

Integration request

The internal interface of a WebSocket API route or REST API method in API Gateway, in which you map the body of a route request or the parameters and body of a method request to the formats required by the backend.

Integration response

The internal interface of a WebSocket API route or REST API method in API Gateway, in which you map the status codes, headers, and payload that are received from the backend to the response format that is returned to a client app.

Mapping template

A script in Velocity Template Language (VTL) that transforms a request body from the frontend data format to the backend data format, or that transforms a response body from the backend data format to the frontend data format. Mapping templates can be specified in the integration request or in the integration response. They can reference data made available at runtime as context and stage variables.

The mapping can be as simple as an identity transform that passes the headers or body through the integration as-is from the client to the backend for a request. The same is true for a response, in which the payload is passed from the backend to the client.

Method request

The public interface of an API method in API Gateway that defines the parameters and body that an app developer must send in requests to access the backend through the API.

Method response

The public interface of a REST API that defines the status codes, headers, and body models that an app developer should expect in responses from the API.

Mock integration

In a mock integration, API responses are generated from API Gateway directly, without the need for an integration backend. As an API developer, you decide how API Gateway responds to a mock integration request. For this, you configure the method's integration request and integration response to associate a response with a given status code.

Model

A data schema specifying the data structure of a request or response payload. A model is required for generating a strongly typed SDK of an API. It is also used to validate payloads. A model is convenient for generating a sample mapping template to initiate creation of a production mapping template. Although useful, a model is not required for creating a mapping template.

Private API

See Private API endpoint.

Private API endpoint

An API endpoint that is exposed through interface VPC endpoints and allows a client to securely access private API resources inside a VPC. Private APIs are isolated from the public internet, and they can only be accessed using VPC endpoints for API Gateway that have been granted access.

Private integration

An API Gateway integration type for a client to access resources inside a customer's VPC through a private REST API endpoint without exposing the resources to the public internet.

Proxy integration

A simplified API Gateway integration configuration. You can set up a proxy integration as an HTTP proxy integration or a Lambda proxy integration.

For HTTP proxy integration, API Gateway passes the entire request and response between the frontend and an HTTP backend. For Lambda proxy integration, API Gateway sends the entire request as input to a backend Lambda function. API Gateway then transforms the Lambda function output to a frontend HTTP response.

In REST APIs, proxy integration is most commonly used with a proxy resource, which is represented by a greedy path variable (for example, `{proxy+}`) combined with a catch-all `ANY` method.

Quick create

You can use quick create to simplify creating an HTTP API. Quick create creates an API with a Lambda or HTTP integration, a default catch-all route, and a default stage that is configured to automatically deploy changes. For more information, see the section called “Create an HTTP API by using the AWS CLI”.

Regional API endpoint

The host name of an API that is deployed to the specified Region and intended to serve clients, such as EC2 instances, in the same AWS Region. API requests are targeted directly to the Region-specific API Gateway API without going through any CloudFront distribution. For in-Region requests, a Regional endpoint bypasses the unnecessary round trip to a CloudFront distribution.

In addition, you can apply latency-based routing on Regional endpoints to deploy an API to multiple Regions using the same Regional API endpoint configuration, set the same custom domain name for each deployed API, and configure latency-based DNS records in Route 53 to route client requests to the Region that has the lowest latency.

See API endpoints.

Route

A WebSocket route in API Gateway is used to direct incoming messages to a specific integration, such as an AWS Lambda function, based on the content of the message. When you define your WebSocket API, you specify a route key and an integration backend. The route key is an attribute in the message body. When the route key is matched in an incoming message, the integration backend is invoked.

A default route can also be set for non-matching route keys or to specify a proxy model that passes the message through as-is to backend components that perform the routing and process the request.

Route request

The public interface of a WebSocket API method in API Gateway that defines the body that an app developer must send in the requests to access the backend through the API.

Route response

The public interface of a WebSocket API that defines the status codes, headers, and body models that an app developer should expect from API Gateway.

Usage plan

A usage plan provides selected API clients with access to one or more deployed REST or WebSocket APIs. You can use a usage plan to configure throttling and quota limits, which are enforced on individual client API keys.

WebSocket connection

API Gateway maintains a persistent connection between clients and API Gateway itself. There is no persistent connection between API Gateway and backend integrations such as Lambda functions. Backend services are invoked as needed, based on the content of messages received from clients.

Choosing between REST APIs and HTTP APIs

REST APIs and HTTP APIs are both RESTful API products. REST APIs support more features than HTTP APIs, while HTTP APIs are designed with minimal features so that they can be offered at a lower price. Choose REST APIs if you need features such as API keys, per-client throttling, request validation, AWS WAF integration, or private API endpoints. Choose HTTP APIs if you don't need the features included with REST APIs.

The following sections summarize core features that are available in REST APIs and HTTP APIs.

Endpoint type

The endpoint type refers to the endpoint that API Gateway creates for your API. For more information, see the section called “Choose an API endpoint type”.

Endpoint types	REST API	HTTP API
Edge-optimized	✓	
Regional	✓	✓
Private	✓	

Security

API Gateway provides a number of ways to protect your API from certain threats, like malicious actors or spikes in traffic. To learn more, see the section called "Protect" and the section called "Protect".

Security features	REST API	HTTP API
Mutual TLS authentication	✓	✓
Certificates for backend authentication	✓	
AWS WAF	✓	

Authorization

API Gateway supports multiple mechanisms for controlling and managing access to your API. For more information, see the section called "Access control" and the section called "Access control".

Authorization options	REST API	HTTP API
IAM	✓	✓
Resource policies	✓	
Amazon Cognito	✓	✓ [1]
Custom authorization with an AWS Lambda function	✓	✓
JSON Web Token (JWT) [2]		✓

[1] You can use Amazon Cognito with a JWT authorizer.

[2] You can use a Lambda authorizer to validate JWTs for REST APIs.

API management

Choose REST APIs if you need API management capabilities such as API keys and per-client rate limiting. For more information, see the section called “Distribute”, the section called “Custom domain names”, and the section called “Custom domain names”.

Features	REST API	HTTP API
Custom domains	✓	✓
API keys	✓	
Per-client rate limiting	✓	
Per-client usage throttling	✓	

Development

As you're developing your API Gateway API, you decide on a number of characteristics of your API. These characteristics depend on the use case of your API. For more information see the section called “Develop” and the section called “Develop”.

Features	REST API	HTTP API
CORS configuration	✓	✓
Test invocations	✓	
Caching	✓	
User-controlled deployments	✓	✓
Automatic deployments		✓
Custom gateway responses	✓	
Canary release deployments	✓	
Request validation	✓	

Features	REST API	HTTP API
Request parameter transformation	✓	✓
Request body transformation	✓	

Monitoring

API Gateway supports several options to log API requests and monitor your APIs. For more information, see the section called "Monitor" and the section called "Monitor".

Feature	REST API	HTTP API
Amazon CloudWatch metrics	✓	✓
Access logs to CloudWatch Logs	✓	✓
Access logs to Amazon Data Firehose	✓	
Execution logs	✓	
AWS X-Ray tracing	✓	

Integrations

Integrations connect your API Gateway API to backend resources. For more information, see the section called "Integrations" and the section called "Integrations".

Feature	REST API	HTTP API
Public HTTP endpoints	✓	✓
AWS services	✓	✓

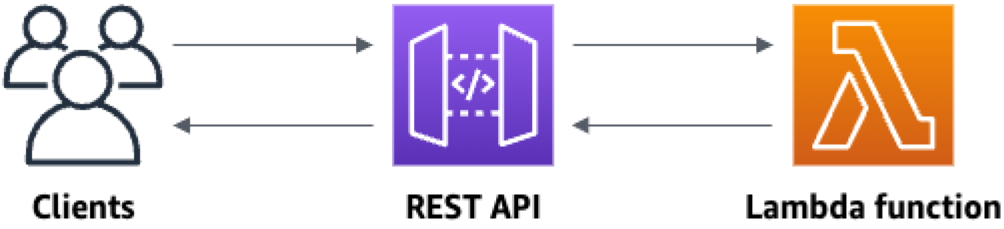
Feature	REST API	HTTP API
AWS Lambda functions	✓	✓
Private integrations with Network Load Balancers	✓	✓
Private integrations with Application Load Balancers		✓
Private integrations with AWS Cloud Map		✓
Mock integrations	✓	

Getting started with the REST API console

In this getting started exercise, you create a serverless REST API using the API Gateway REST API console. Serverless APIs let you focus on your applications instead of spending your time provisioning and managing servers. This exercise should take less than 20 minutes to complete, and is possible within the AWS Free Tier.

First, you create a Lambda function using the Lambda console. Next, you create a REST API using the API Gateway REST API console. Then, you create an API method and integrate it with a Lambda function using a Lambda proxy integration. Finally, you deploy and invoke your API.

When you invoke your REST API, API Gateway routes the request to your Lambda function. Lambda runs the function and returns a response to API Gateway. API Gateway then returns that response to you.



To complete this exercise, you need an AWS account and an AWS Identity and Access Management (IAM) user with console access. For more information, see Prerequisites for getting started with API Gateway.

Topics

Step 1: Create a Lambda function

You use a Lambda function for the backend of your API. Lambda runs your code only when needed and scales automatically, from a few requests per day to thousands per second.

For this exercise, you use a default Node.js function in the Lambda console.

To create a Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. Choose **Create function**.
3. Under **Basic information**, for **Function name**, enter **`my-function`**.
4. Choose **Create function**.

The default Lambda function code should look similar to the following:

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('The API Gateway REST API console is great!'),
    };
    return response;
};
```

You can modify your Lambda function for this exercise, as long as the function's response aligns with the format that API Gateway requires.

Replace the default response body (`Hello from Lambda!`) with `The API Gateway REST API console is great!`. When you invoke the example function, it returns a `200` response to clients, along with the updated response.

Step 2: Create a REST API

Next, you create a REST API with a root resource (/).

To create a REST API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
   - To create your first API, for **REST API**, choose **Build**.
   - If you've created an API before, choose **Create API**, and then choose **Build** for **REST API**.
3. For **API name**, enter **`my-rest-api`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

Step 3: Create a Lambda proxy integration

Next, you create an API method for your REST API on the root resource (/) and integrate the method with your Lambda function using a proxy integration. In a Lambda proxy integration, API Gateway passes the incoming request from the client directly to the Lambda function.

To create a Lambda proxy integration

1. Select the / resource, and then choose **Create method**.
2. For **Method type**, select ANY.
3. For **Integration type**, select **Lambda**.
4. Turn on **Lambda proxy integration**.
5. For **Lambda function**, enter **`my-function`**, and then select your Lambda function.
6. Choose **Create method**.

Step 4: Deploy your API

Next, you create an API deployment and associate it with a stage.

To deploy your API

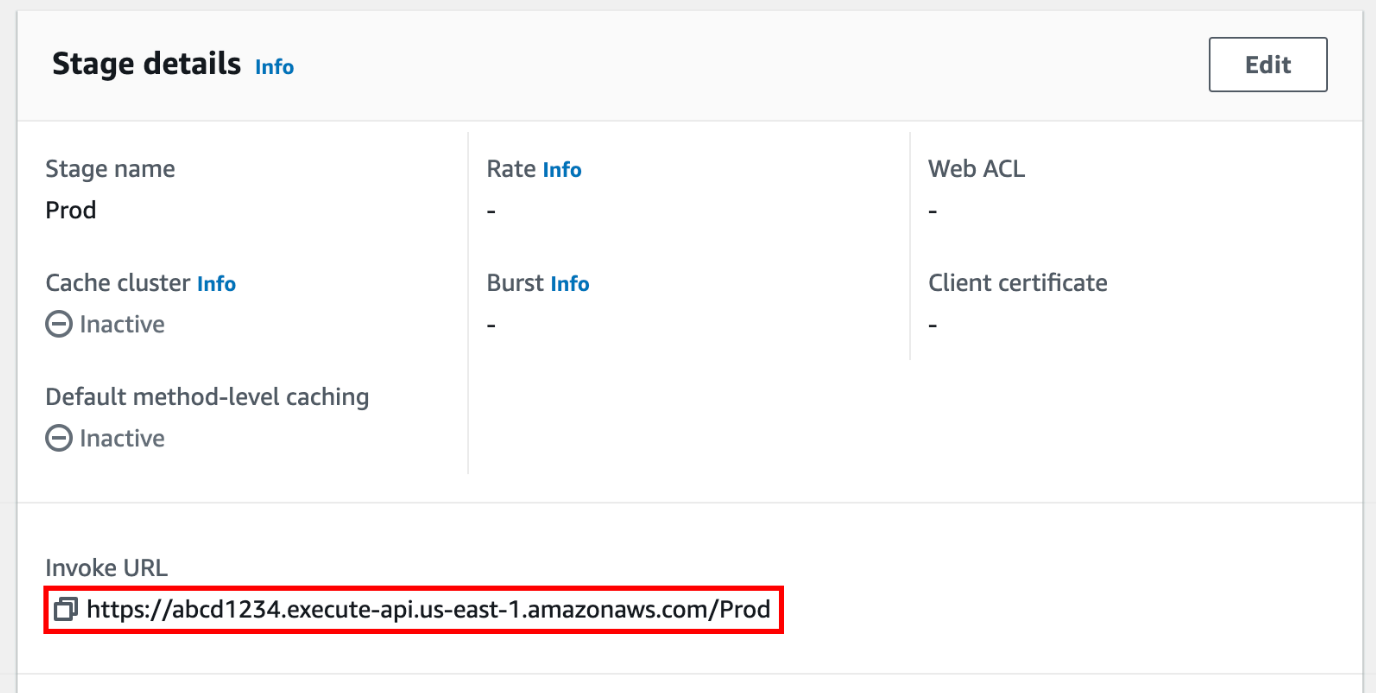
1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`Prod`**.
4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

Now clients can call your API. To test your API before deploying it, you can optionally choose the **ANY** method, navigate to the **Test** tab, and then choose **Test**.

Step 5: Invoke your API

To invoke your API

1. From the main navigation pane, choose **Stage**.
2. Under **Stage details**, choose the copy icon to copy your API's invoke URL.



Stage details Info

Stage name
Prod

Cache cluster Info
⊖ Inactive

Default method-level caching
⊖ Inactive

Rate Info
-

Burst Info
-

Web ACL
-

Client certificate
-

Invoke URL
https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod

3. Enter the invoke URL in a web browser.

The full URL should look like `https://abcd123.execute-api.us-east-2.amazonaws.com/Prod`.

Your browser sends a `GET` request to the API.

4. Verify your API's response. You should see the text `"The API Gateway REST API console is great!"` in your browser.

(Optional) Step 6: Clean up

To prevent accruing unnecessary costs to your AWS account, delete the resources that you created as part of this exercise. The following steps delete your REST API, your Lambda function, and the associated resources.

To delete your REST API

1. In the **Resources** pane, choose **API actions**, **Delete API**.
2. In the **Delete API** dialog box, enter **confirm**, and then choose **Delete**.

To delete your Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. On the **Functions** page, select your function. Choose **Actions**, **Delete**.
3. In the **Delete 1 functions** dialog box, enter **`delete`**, and then choose **Delete**.

To delete your Lambda function's log group

1. Open the Log groups page of the Amazon CloudWatch console.
2. On the **Log groups** page, select your function's log group (`/aws/lambda/my-function`). Then, for **Actions**, choose **Delete log group(s)**.
3. In the **Delete log group(s)** dialog box, choose **Delete**.

To delete your Lambda function's execution role

1. Open the Roles page of the IAM console.
2. (Optional) On the **Roles** page, in the search box, enter **`my-function`**.

3. Select your function's role (for example, `my-function-31exxmpl`), and then choose **Delete**.
4. In the **Delete `my-function-31exxmpl`?** dialog box, enter the name of the role, and then choose **Delete**.

> **Tip**
>
> You can automate the creation and cleanup of AWS resources by using AWS CloudFormation or AWS Serverless Application Model (AWS SAM). For some example AWS CloudFormation templates, see the example templates for API Gateway in the **awsdocs** GitHub repository.

Prerequisites for getting started with API Gateway

Before you use Amazon API Gateway for the first time, complete the following tasks.

Sign up for an AWS account

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

To sign up for an AWS account

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

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

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

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

Create an administrative user

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

Secure your AWS account root user

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

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

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

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

Create an administrative user

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.
2. In IAM Identity Center, grant administrative access to an administrative user.

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

Sign in as the administrative user

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

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

Getting started with API Gateway

Note

The redesigned API Gateway console experience is now available. For a tutorial on how to use the console to create a REST API, see Getting started with the REST API console.

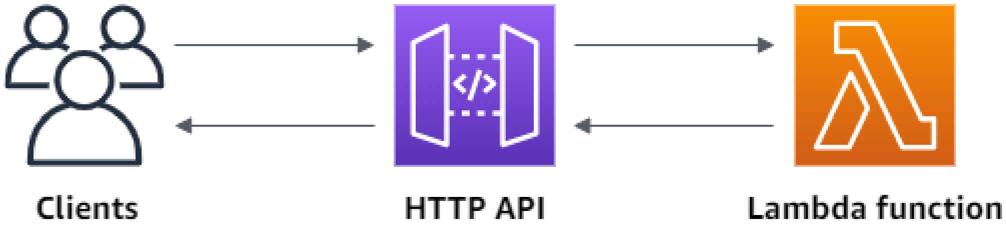
In this getting started exercise, you create a serverless API. Serverless APIs let you focus on your applications, instead of spending time provisioning and managing servers. This exercise takes less than 20 minutes to complete, and is possible within the AWS Free Tier.

First, you create a Lambda function using the AWS Lambda console. Next, you create an HTTP API using the API Gateway console. Then, you invoke your API.

Note

This exercise uses an HTTP API for simplicity. API Gateway also supports REST APIs, which include more features. To learn more, see the section called "Choosing between REST APIs and HTTP APIs ".

When you invoke your HTTP API, API Gateway routes the request to your Lambda function. Lambda runs the Lambda function and returns a response to API Gateway. API Gateway then returns a response to you.



To complete this exercise, you need an AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.

Topics

Step 1: Create a Lambda function

You use a Lambda function for the backend of your API. Lambda runs your code only when needed and scales automatically, from a few requests per day to thousands per second.

For this example, you use the default Node.js function from the Lambda console.

To create a Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. Choose **Create function**.
3. For **Function name**, enter **`my-function`**.
4. Choose **Create function**.

The example function returns a `200` response to clients, and the text `Hello from Lambda!`.

You can modify your Lambda function, as long as the function's response aligns with the format that API Gateway requires.

The default Lambda function code should look similar to the following:

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

Step 2: Create an HTTP API

Next, you create an HTTP API. API Gateway also supports REST APIs and WebSocket APIs, but an HTTP API is the best choice for this exercise. REST APIs support more features than HTTP APIs, but we don't need those features for this exercise. HTTP APIs are designed with minimal features so that they can be offered at a lower price. WebSocket APIs maintain persistent connections with clients for full-duplex communication, which isn't required for this example.

The HTTP API provides an HTTP endpoint for your Lambda function. API Gateway routes requests to your Lambda function, and then returns the function's response to clients.

To create an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
   - To create your first API, for **HTTP API**, choose **Build**.
   - If you've created an API before, choose **Create API**, and then choose **Build** for **HTTP API**.
3. For **Integrations**, choose **Add integration**.
4. Choose **Lambda**.
5. For **Lambda function**, enter **`my-function`**.
6. For **API name**, enter **`my-http-api`**.
7. Choose **Next**.
8. Review the *route* that API Gateway creates for you, and then choose **Next**.
9. Review the *stage* that API Gateway creates for you, and then choose **Next**.
10. Choose **Create**.

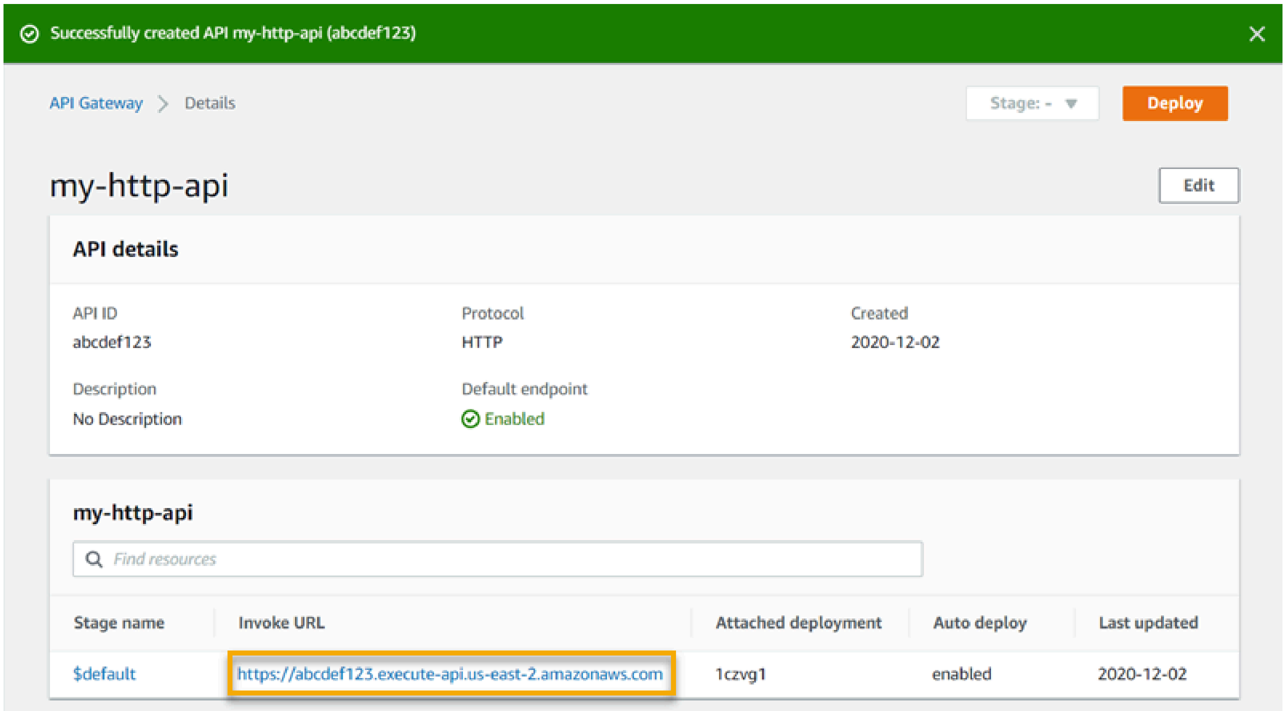
Now you've created an HTTP API with a Lambda integration that's ready to receive requests from clients.

Step 3: Test your API

Next, you test your API to make sure that it's working. For simplicity, use a web browser to invoke your API.

To test your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Note your API's invoke URL.



Successfully created API my-http-api (abcdef123)

my-http-api

API details

API ID: abcdef123
Protocol: HTTP
Created: 2020-12-02
Description: No Description
Default endpoint: Enabled

my-http-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-east-2.amazonaws.com	1czvg1	enabled	2020-12-02

4. Copy your API's invoke URL, and enter it in a web browser. Append the name of your Lambda function to your invoke URL to call your Lambda function. By default, the API Gateway console creates a route with the same name as your Lambda function, `my-function`.

   The full URL should look like `https://`*`abcdef123`*`.execute-api.`*`us-east-2`*`.amazonaws.com/`*`my-function`*.

   Your browser sends a `GET` request to the API.

5. Verify your API's response. You should see the text `"Hello from Lambda!"` in your browser.

(Optional) Step 4: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this getting started exercise. The following steps delete your HTTP API, your Lambda function, and associated resources.

To delete an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select an API. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. On the **Functions** page, select a function. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function's log group

1. In the Amazon CloudWatch console, open the Log groups page.
2. On the **Log groups** page, select the function's log group (`/aws/lambda/my-function`). Choose **Actions**, and then choose **Delete log group**.
3. Choose **Delete**.

To delete a Lambda function's execution role

1. In the AWS Identity and Access Management console, open the Roles page.
2. Select the function's role, for example, `my-function-`*`31exxmpl`*.
3. Choose **Delete role**.
4. Choose **Yes, delete**.

You can automate the creation and cleanup of AWS resources by using AWS CloudFormation or AWS SAM. For example AWS CloudFormation templates, see example AWS CloudFormation templates.

Next steps

For this example, you used the AWS Management Console to create a simple HTTP API. The HTTP API invokes a Lambda function and returns a response to clients.

The following are next steps as you continue to work with API Gateway.

- Configure additional types of API integrations, including:
  - HTTP endpoints
  - Private resources in a VPC, such as Amazon ECS services

  - AWS services such as Amazon Simple Queue Service, AWS Step Functions, and Kinesis Data Streams
- Control access to your APIs
- Enable logging for your APIs
- Configure throttling for your APIs
- Configure custom domains for your APIs

To get help with Amazon API Gateway from the community, see the API Gateway Discussion Forum. When you enter this forum, AWS might require you to sign in.

To get help with API Gateway directly from AWS, see the support options on the AWS Support page.

See also our frequently asked questions (FAQs), or contact us directly.

Amazon API Gateway tutorials and workshops

The following tutorials and workshops provide hands-on exercises to help you learn about API Gateway.

REST API tutorials

- Build an API Gateway REST API with Lambda integration
- Tutorial: Create a REST API by importing an example
- Build an API Gateway REST API with HTTP integration
- Tutorial: Build a REST API with API Gateway private integration
- Tutorial: Build an API Gateway REST API with AWS integration
- Tutorial: Create a `Calc` REST API with two AWS service integrations and one Lambda non-proxy integration
- Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway
- Tutorial: Create a REST API as an Amazon Kinesis proxy in API Gateway
- Tutorial: Building a private REST API

HTTP API tutorials

- Tutorial: Build a CRUD API with Lambda and DynamoDB
- Tutorial: Building an HTTP API with a private integration to an Amazon ECS service

WebSocket API tutorials

- Tutorial: Building a serverless chat app with a WebSocket API, Lambda and DynamoDB

Workshops

- Build a serverless web application
- CI/CD for serverless applications
- Serverless security workshop
- Serverless identity management, authentication and authorization
- The Amazon API Gateway Workshop

Amazon API Gateway REST API tutorials

The following tutorials provide hands-on exercises to help you learn about API Gateway REST APIs.

Topics

Build an API Gateway REST API with Lambda integration

To build an API with Lambda integrations, you can use Lambda proxy integration or Lambda non-proxy integration.

In Lambda proxy integration, the input to the integrated Lambda function can be expressed as any combination of request headers, path variables, query string parameters, and body. In addition, the Lambda function can use API configuration settings to influence its execution logic. For an API developer, setting up a Lambda proxy integration is simple. Other than choosing a particular Lambda function in a given region, you have little else to do. API Gateway configures the integration request and integration response for you. Once set up, the integrated API method can evolve with the backend without modifying the existing settings. This is possible because the backend Lambda function developer parses the incoming request data and responds with desired results to the client when nothing goes wrong or responds with error messages when anything goes wrong.

In Lambda non-proxy integration, you must ensure that input to the Lambda function is supplied as the integration request payload. This implies that you, as an API developer, must map any input data the client supplied as request parameters into the proper integration request body. You might also need to translate the client-supplied request body into a format recognized by the Lambda function.

Topics

- Tutorial: Build a Hello World REST API with Lambda proxy integration
- Tutorial: Build an API Gateway REST API with cross-account Lambda proxy integration
- Tutorial: Build an API Gateway REST API with Lambda non-proxy integration

Tutorial: Build a Hello World REST API with Lambda proxy integration

Lambda proxy integration is a lightweight, flexible API Gateway API integration type that allows you to integrate an API method – or an entire API – with a Lambda function. The Lambda function can be written in any language that Lambda supports. Because it's a proxy integration, you can change the Lambda function implementation at any time without needing to redeploy your API.

In this tutorial, you do the following:

- Create a "Hello, World!" Lambda function to be the backend for the API.
- Create and test a "Hello, World!" API with Lambda proxy integration.

Topics

- Create a "Hello, World!" Lambda function
- Create a "Hello, World!" API
- Deploy and test the API

Create a "Hello, World!" Lambda function

To create a "Hello, World!" Lambda function in the Lambda console

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. On the AWS navigation bar, choose a Region (for example, US East (N. Virginia)).

   > **Note**
   >
   > Note the region where you create the Lambda function. You'll need it when you create the API.

3. Choose **Functions** in the navigation pane.
4. Choose **Create function**.
5. Choose **Author from scratch**.
6. Under **Basic information**, do the following:

    a. In **Function name**, enter **`GetStartedLambdaProxyIntegration`**.

    b. For **Runtime**, choose either the latest supported **Node.js** or **Python** runtime.

    c. Under **Permissions**, expand **Change default execution role**. For **Execution role** dropdown list, choose **Create new role from AWS policy templates**.

    d. In **Role name**, enter **`GetStartedLambdaBasicExecutionRole`**.

    e. Leave the **Policy templates** field blank.

    f. Choose **Create function**.

7. Under **Function code**, in the inline code editor, copy/paste the following code:

    Node.js

```
export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    var greeter = 'World';
    if (event.greeter && event.greeter!=="") {
        greeter =  event.greeter;
    } else if (event.body && event.body !== "") {
        var body = JSON.parse(event.body);
        if (body.greeter && body.greeter !== "") {
            greeter = body.greeter;
        }
    } else if (event.queryStringParameters &&
 event.queryStringParameters.greeter && event.queryStringParameters.greeter !==
 "") {
        greeter = event.queryStringParameters.greeter;
    } else if (event.multiValueHeaders && event.multiValueHeaders.greeter &&
 event.multiValueHeaders.greeter != "") {
        greeter = event.multiValueHeaders.greeter.join(" and ");
    } else if (event.headers && event.headers.greeter && event.headers.greeter !
= "") {
        greeter = event.headers.greeter;
    }

    res.body = "Hello, " + greeter + "!";
    callback(null, res);
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)

    greeter = 'World'

    try:
        if (event['queryStringParameters']) and (event['queryStringParameters']
['greeter']) and (
                event['queryStringParameters']['greeter'] is not None):
            greeter = event['queryStringParameters']['greeter']
    except KeyError:
        print('No greeter')

    try:
        if (event['multiValueHeaders']) and (event['multiValueHeaders']
['greeter']) and (
                event['multiValueHeaders']['greeter'] is not None):
            greeter = " and ".join(event['multiValueHeaders']['greeter'])
    except KeyError:
        print('No greeter')

    try:
        if (event['headers']) and (event['headers']['greeter']) and (
                event['headers']['greeter'] is not None):
            greeter = event['headers']['greeter']
    except KeyError:
        print('No greeter')

    if (event['body']) and (event['body'] is not None):
        body = json.loads(event['body'])
        try:
            if (body['greeter']) and (body['greeter'] is not None):
                greeter = body['greeter']
        except KeyError:
            print('No greeter')

    res = {
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        },
        "body": "Hello, " + greeter + "!"
    }

    return res
```

8. Choose **Deploy**.

Create a "Hello, World!" API

Now create an API for your "Hello, World!" Lambda function by using the API Gateway console.

To create a "Hello, World!" API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. For **API name**, enter **`LambdaProxyAPI`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

After you create an API, you create a resource. Typically, API resources are organized in a resource tree according to the application logic. For this example, you create a **/helloworld** resource.

To create a resource

1. Select the **/** resource, and then choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. Keep **Resource path** as `/`.
4. For **Resource name**, enter **`helloworld`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

In a proxy integration, the entire request is sent to the backend Lambda function as-is, via a catch-all `ANY` method that represents any HTTP method. The actual HTTP method is specified by the client at run time. The `ANY` method allows you to use a single API method setup for all of the supported HTTP methods: `DELETE`, `GET`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, and `PUT`.

To create an `ANY` method

1. Select the **/helloworld** resource, and then choose **Create method**.
2. For **Method type**, select **ANY**.
3. For **Integration type**, select **Lambda function**.
4. Turn on **Lambda proxy integration**.
5. For **Lambda function**, select the AWS Region where you created your Lambda function, and then enter the function name.
6. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.
7. Choose **Create method**.

Deploy and test the API

To deploy your API

1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`test`**.

4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.
6. Under **Stage details**, choose the copy icon to copy your API's invoke URL.

Use browser and cURL to test an API with Lambda proxy integration

You can use a browser or cURL to test your API.

To test GET requests using only query string parameters, you can enter the URL for the API's helloworld resource into a browser address bar.

To create the URL for the API's helloworld resource, append the resource helloworld and the query string parameter ?greeter=John to your invoke URL. Your URL should look like the following.

```
https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/helloworld?greeter=John
```

For other methods, you must use more advanced REST API testing utilities, such as POSTMAN or cURL. This tutorial uses cURL. The cURL command examples below assume that cURL is installed on your computer.

To test your deployed API using cURL:

1. Open a terminal window.
2. Copy the following cURL command and paste it into the terminal window, and replace the invoke URL with the one you copied in the previous step and add **/helloworld** to the end of the URL.

   > **Note**
   >
   > If you're running the command on Windows, use this syntax instead:
   >
   > ```
   > curl -v -X POST "https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/
   > test/helloworld" -H "content-type: application/json" -d "{ \"greeter\":
   >  \"John\" }"
   > ```

   a. To call the API with the query string parameter of ?greeter=John:

```
curl -X GET 'https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
helloworld?greeter=John'
```

b. To call the API with a header parameter of `greeter:John`:

```
curl -X GET https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
helloworld \
  -H 'content-type: application/json' \
  -H 'greeter: John'
```

c. To call the API with a body of `{"greeter":"John"}`:

```
curl -X POST https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
helloworld \
  -H 'content-type: application/json' \
  -d '{ "greeter": "John" }'
```

In all the cases, the output is a 200 response with the following response body:

```
Hello, John!
```

Tutorial: Build an API Gateway REST API with cross-account Lambda proxy integration

You can now use an AWS Lambda function from a different AWS account as your API integration backend. Each account can be in any region where Amazon API Gateway is available. This makes it easy to centrally manage and share Lambda backend functions across multiple APIs.

In this section, we show how to configure cross-account Lambda proxy integration using the Amazon API Gateway console.

Create API for API Gateway cross-account Lambda integration

To create an API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.

2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. For **API name**, enter **`CrossAccountLambdaAPI`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

Create Lambda integration function in another account

Now you'll create a Lambda function in a different account from the one in which you created the example API.

Creating a Lambda function in another account

1. Log in to the Lambda console in a different account from the one where you created your API Gateway API.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Under **Author from scratch**, do the following:

   a. For **Function name**, enter a name.

   b. From the **Runtime** drop-down list, choose a supported Node.js runtime.

   c. Under **Permissions**, expand **Choose or create an execution role**. You can create a role or choose an existing role.

   d. Choose **Create function** to continue.
5. Scroll down to the **Function code** pane.
6. Enter the Node.js function implementation from the section called “Tutorial: Hello World API with Lambda proxy integration”.
7. Choose **Deploy**.
8. Note the full ARN for your function (in the upper right corner of the Lambda function pane). You'll need it when you create your cross-account Lambda integration.

Configure cross-account Lambda integration

Once you have a Lambda integration function in a different account, you can use the API Gateway console to add it to your API in your first account.

> **Note**
>
> If you are configuring a cross-region, cross-account authorizer, the `sourceArn` that is added to the target function should use the region of the function, not the region of the API.

After you create an API, you create a resource. Typically, API resources are organized in a resource tree according to the application logic. For this example, you create a **/helloworld** resource.

To create a resource

1. Select the **/** resource, and then choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. Keep **Resource path** as `/`.
4. For **Resource name**, enter **`helloworld`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

After you create an resource, you create a `GET` method. You integrate the `GET` method with a Lambda function in another account.

To create a GET method

1. Select the **/helloworld** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **Lambda function**.
4. Turn on **Lambda proxy integration**.
5. For **Lambda function**, enter the full ARN of your Lambda function from Step 1.

   In the Lambda console, you can find the ARN for your function in the upper right corner of the console window.

6. When you enter the ARN, a `aws lambda add-permission` command string will appear. This policy grants your first account access to your second account's Lambda function. Copy and paste the `aws lambda add-permission` command string into an AWS CLI window that is configured for your second account.
7. Choose **Create method**.

You can see your updated policy for your function in the Lambda console.

(Optional) To see your updated policy

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose your Lambda function.
3. Choose **Permissions**.

   You should see an `Allow` policy with a `Condition` clause in which the in the `AWS:SourceArn` is the ARN for your API's `GET` method.

Tutorial: Build an API Gateway REST API with Lambda non-proxy integration

In this walkthrough, we use the API Gateway console to build an API that enables a client to call Lambda functions through the Lambda non-proxy integration (also known as custom integration). For more information about AWS Lambda and Lambda functions, see the AWS Lambda Developer Guide.

To facilitate learning, we chose a simple Lambda function with minimal API setup to walk you through the steps of building an API Gateway API with the Lambda custom integration. When necessary, we describe some of the logic. For a more detailed example of the Lambda custom integration, see Tutorial: Create a `Calc` REST API with two AWS service integrations and one Lambda non-proxy integration.

Before creating the API, set up the Lambda backend by creating a Lambda function in AWS Lambda, described next.

Topics

- Test invoking the API method
- Deploy the API
- Test the API in a deployment stage
- Clean up

Create a Lambda function for Lambda non-proxy integration

Note

Creating Lambda functions may result in charges to your AWS account.

In this step, you create a "Hello, World!"-like Lambda function for the Lambda custom integration. Throughout this walkthrough, the function is called `GetStartedLambdaIntegration`.

The implementation of this `GetStartedLambdaIntegration` Lambda function is as follows:

Node.js

```
'use strict';
var days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
 'Saturday'];
var times = ['morning', 'afternoon', 'evening', 'night', 'day'];

console.log('Loading function');

export const handler = function(event, context, callback) {
  // Parse the input for the name, city, time and day property values
  let name = event.name === undefined ? 'you' : event.name;
  let city = event.city === undefined ? 'World' : event.city;
  let time = times.indexOf(event.time)<0 ? 'day' : event.time;
  let day = days.indexOf(event.day)<0 ? null : event.day;

  // Generate a greeting
  let greeting = 'Good ' + time + ', ' + name + ' of ' + city + '. ';
  if (day) greeting += 'Happy ' + day + '!';

  // Log the greeting to CloudWatch
  console.log('Hello: ', greeting);
  // Return a greeting to the caller
  callback(null, {
      "greeting": greeting
  });
};
```

Python

```
import json

days = {
    'Sunday',
    'Monday',
    'Tuesday',
    'Wednesday',
    'Thursday',
    'Friday',
    'Saturday'}
times = {'morning', 'afternoon', 'evening', 'night', 'day'}


def lambda_handler(event, context):
    print(event)
    # parse the input for the name, city, time, and day property values
    try:
        if event['name']:
            name = event['name']
    except KeyError:
        name = 'you'
    try:
        if event['city']:
            city = event['city']
    except KeyError:
        city = 'World'
    try:
        if event['time'] in times:
            time = event['time']
        else:
            time = 'day'
    except KeyError:
        time = 'day'
    try:
        if event['day'] in days:
            day = event['day']
        else:
            day = ''
    except KeyError:
        day = ''
    # Generate a greeting
    greeting = 'Good ' + time + ', ' + name + ' of ' + \
        city + '.' + ['', ' Happy ' + day + '!'][day != '']
    # Log the greeting to CloudWatch
    print(greeting)

    # Return a greeting to the caller
    return {"greeting": greeting}
```

For the Lambda custom integration, API Gateway passes the input to the Lambda function from the client as the integration request body. The `event` object of the Lambda function handler is the input.

Our Lambda function is simple. It parses the input `event` object for the `name`, `city`, `time`, and `day` properties. It then returns a greeting, as a JSON object of `{"message":greeting}`, to the caller. The message is in the `"Good [morning|afternoon|day], [`*`name`*`|you] in [`*`city`*`|World]. Happy `*`day`*`!"` pattern. It is assumed that the input to the Lambda function is of the following JSON object:

```
{
  "city": "...",
  "time": "...",
  "day": "...",
  "name" : "..."
}
```

For more information, see the [AWS Lambda Developer Guide](https://docs.aws.amazon.com/lambda/latest/dg/).

In addition, the function logs its execution to Amazon CloudWatch by calling `console.log(...)`. This is helpful for tracing calls when debugging the function. To allow the `GetStartedLambdaIntegration` function to log the call, set an IAM role with appropriate policies for the Lambda function to create the CloudWatch streams and add log entries to the streams. The Lambda console guides you through to create the required IAM roles and policies.

If you set up the API without using the API Gateway console, such as when importing an API from an OpenAPI file, you must explicitly create, if necessary, and set up an invocation role and policy for API Gateway to invoke the Lambda functions. For more information on how to set up Lambda invocation and execution roles for an API Gateway API, see Control access to an API with IAM permissions.

Compared to `GetStartedLambdaProxyIntegration`, the Lambda function for the Lambda proxy integration, the `GetStartedLambdaIntegration` Lambda function for the Lambda custom integration only takes input from the API Gateway API integration request body. The function can return an output of any JSON object, a string, a number, a Boolean, or even a binary blob. The Lambda function for the Lambda proxy integration, in contrast, can take the input from any request data, but must return an output of a particular JSON object. The `GetStartedLambdaIntegration` function for the Lambda custom integration can have the API request parameters as input, provided that API Gateway maps the required API request parameters to the integration request body before forwarding the client request to the backend. For this to happen, the API developer must create a mapping template and configure it on the API method when creating the API.

Now, create the `GetStartedLambdaIntegration` Lambda function.

To create the `GetStartedLambdaIntegration` Lambda function for Lambda custom integration

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Do one of the following:
   - If the welcome page appears, choose **Get Started Now** and then choose **Create function**.
   - If the **Lambda > Functions** list page appears, choose **Create function**.
3. Choose **Author from scratch**.
4. In the **Author from scratch** pane, do the following:
   a. For **Name**, enter **`GetStartedLambdaIntegration`** as the Lambda function name.
   b. For **Runtime**, choose either the latest supported **Node.js** or **Python** runtime.
   c. Under **Permissions**, expand **Change default execution role**. For **Execution role** dropdown list, choose **Create new role from AWS policy templates**.
   d. For **Role name**, enter a name for your role (for example, **`GetStartedLambdaIntegrationRole`**).

    e. For **Policy templates**, choose **Simple microservice permissions**.

    f. Choose **Create function**.

5. In the **Configure function** pane, under **Function code** do the following:

    a. Copy the Lambda function code listed in the beginning of this section and paste it in the inline code editor.

    b. Leave the default choices for all other fields in this section.

    c. Choose **Deploy**.

6. To test the newly created function, choose the **Test** tab.

    a. For **Event name**, enter **`HelloWorldTest`**.

    b. For **Event JSON**, replace the default code with the following.

    ```
    {
      "name": "Jonny",
      "city": "Seattle",
      "time": "morning",
      "day": "Wednesday"
    }
    ```

    c. Choose **Test** to invoke the function. The **Execution result: succeeded** section is shown. Expand **Details** and you see the following output.

    ```
    {
        "greeting": "Good morning, Jonny of Seattle. Happy Wednesday!"
    }
    ```

    The output is also written to CloudWatch Logs.

As a side exercise, you can use the IAM console to view the IAM role (`GetStartedLambdaIntegrationRole`) that was created as part of the Lambda function creation. Attached to this IAM role are two inline policies. One stipulates the most basic permissions for Lambda execution. It permits calling the CloudWatch `CreateLogGroup` for any CloudWatch resources of your account in the region where the Lambda function is created. This policy also allows creating the CloudWatch streams and logging events for the `HelloWorldForLambdaIntegration` Lambda function.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "logs:CreateLogGroup",
            "Resource": "arn:aws:logs:region:account-id:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:region:account-id:log-group:/aws/lambda/
GetStartedLambdaIntegration:*"
            ]
        }
    ]
}
```

The other policy document applies to invoking another AWS service that is not used in this example. You can skip it for now.

Associated with the IAM role is a trusted entity, which is `lambda.amazonaws.com`. Here is the trust relationship:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

The combination of this trust relationship and the inline policy makes it possible for the Lambda function to invoke a `console.log()` function to log events to CloudWatch Logs.

If you did not use the AWS Management Console to create the Lambda function, you need to follow these examples to create the required IAM role and policies and then manually attach the role to your function.

Create an API with Lambda non-proxy integration

With the Lambda function (`GetStartedLambdaIntegration`) created and tested, you are ready to expose the function through an API Gateway API. For illustration purposes, we expose the Lambda function with a generic HTTP method. We use the request body, a URL path variable, a query string, and a header to receive required input data from the client. We turn on the API Gateway request validator for the API to ensure that all of the required data is properly defined and specified. We configure a mapping template for API Gateway to transform the client-supplied request data into the valid format as required by the backend Lambda function.

To create an API with a Lambda non-proxy integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. For **API name**, enter **`LambdaNonProxyAPI`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

After creating your API, you create a **/{city}** resource. This is an example of a resource with a path variable that takes an input from the client. Later, you map this path variable into the Lambda function input using a mapping template.

To create a resource

1. Choose **Create resource**.

2. Keep **Proxy resource** turned off.
3. Keep **Resource path** as `/`.
4. For **Resource name**, enter **`{city}`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

After creating your **/{city}** resource, you create an `ANY` method. The `ANY` HTTP verb is a placeholder for a valid HTTP method that a client submits at run time. This example shows that `ANY` method can be used for Lambda custom integration as well as for Lambda proxy integration.

To create an `ANY` method

1. Select the **/{city}** resource, and then choose **Create method**.
2. For **Method type**, select **ANY**.
3. For **Integration type**, select **Lambda function**.
4. Keep **Lambda proxy integration** turned off.
5. For **Lambda function**, select the AWS Region where you created your Lambda function, and then enter the function name.
6. Choose **Method request settings.**

   Now, you turn on a request validator for a URL path variable, a query string parameter, and a header to ensure that all of the required data is defined. For this example, you create a `time` query string parameter and a `day` header.
7. For **Request validator**, select **Validate query string parameters and headers**.
8. Choose **URL query string parameters** and do the following:

   a. Choose **Add query string**.
   b. For **Name**, enter **`time`**.
   c. Turn on **Required**.
   d. Keep **Caching** turned off.
9. Choose **HTTP request headers** and do the following:

   a. Choose **Add header**.
   b. For **Name**, enter **`day`**.

   c. Turn on **Required**.
   d. Keep **Caching** turned off.

10. Choose **Create method**.

After turning on a request validator, you configure the integration request for the ANY method by adding a body-mapping template to transform the incoming request into a JSON payload, as required by the backend Lambda function.

To configure the integration request

1. On the **Integration request** tab, under the **Integration request settings**, choose **Edit**.
2. For **Request body passthrough**, select **When there are no templates defined (recommended)**.
3. Choose **Mapping templates**.
4. Choose **Add mapping template**.
5. For **Content type**, enter **`application/json`**.
6. For **Template body**, enter the following code:

```
#set($inputRoot = $input.path('$'))
{
  "city": "$input.params('city')",
  "time": "$input.params('time')",
  "day":  "$input.params('day')",
  "name": "$inputRoot.callerName"
}
```

7. Choose **Save**.

Test invoking the API method

The API Gateway console provides a testing facility for you to test invoking the API before it is deployed. You use the Test feature of the console to test the API by submitting the following request:

```
POST /Seattle?time=morning
day:Wednesday

{
    "callerName": "John"
}
```

In this test request, you'll set ANY to POST, set {city} to Seattle, assign Wednesday as the day header value, and assign "John" as the callerName value.

To test the ANY method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Method type**, select POST.
3. For **Path**, under **city**, enter **Seattle**.
4. For **Query strings**, enter **time=morning**.
5. For **Headers**, enter **day:Wednesday**.
6. For **Request Body**, enter **{ "callerName": "John" }**.
7. Choose **Test**.

Verify that the returned response payload is as follows:

```
{
  "greeting": "Good morning, John of Seattle. Happy Wednesday!"
}
```

You can also view the logs to examine how API Gateway processes the request and response.

```
Execution log for request test-request
Thu Aug 31 01:07:25 UTC 2017 : Starting execution for request: test-invoke-request
Thu Aug 31 01:07:25 UTC 2017 : HTTP Method: POST, Resource Path: /Seattle
Thu Aug 31 01:07:25 UTC 2017 : Method request path: {city=Seattle}
Thu Aug 31 01:07:25 UTC 2017 : Method request query string: {time=morning}
Thu Aug 31 01:07:25 UTC 2017 : Method request headers: {day=Wednesday}
Thu Aug 31 01:07:25 UTC 2017 : Method request body before transformations:
 { "callerName": "John" }
Thu Aug 31 01:07:25 UTC 2017 : Request validation succeeded for content type
 application/json
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request URI: https://
lambda.us-west-2.amazonaws.com/2015-03-31/functions/arn:aws:lambda:us-
west-2:123456789012:function:GetStartedLambdaIntegration/invocations
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request headers: {x-amzn-lambda-integration-
tag=test-request,
 Authorization=*******************************************************************************
 X-Amz-Date=20170831T010725Z, x-amzn-apigateway-api-id=beags1mnid, X-Amz-
Source-Arn=arn:aws:execute-api:us-west-2:123456789012:beags1mnid/null/POST/
{city}, Accept=application/json, User-Agent=AmazonAPIGateway_beags1mnid,
 X-Amz-Security-Token=FQoDYXdzELL//////////wEaDMHGzEdEOT/VvGhabiK3AzgKrJw
+3zLqJZG4PhOq12K6W21+QotY2rrZyOzqhLoiuRg3CAYNQ2eqgL5D54+63ey9bIdtwHGoyBdq8ecWxJK/
YUnT2Rau0L9HCG5p7FC05h3IvwlFfvcidQNXeYvsKJTLXI05/
yEnY3ttIAnpNYLOezD9Es8rBfyruHfJfOqextKlsC8DymCcqlGkig8qLKcZ0hWJWVwiPJiFgL7laabXs+
+ZhCa4hdZo4iqlG729DE4gaV1mJVdoAagIUwLMo+y4NxFDu0r7I0/
EO5nYcCrppGVVBYiGk7H4T6sXuhTkbNNqVmXtV3ch5bOlh7 [TRUNCATED]
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request body after transformations: {
  "city": "Seattle",
  "time": "morning",
  "day": "Wednesday",
  "name" : "John"
}
Thu Aug 31 01:07:25 UTC 2017 : Sending request to https://lambda.us-
west-2.amazonaws.com/2015-03-31/functions/arn:aws:lambda:us-
west-2:123456789012:function:GetStartedLambdaIntegration/invocations
Thu Aug 31 01:07:25 UTC 2017 : Received response. Integration latency: 328 ms
Thu Aug 31 01:07:25 UTC 2017 : Endpoint response body before transformations:
 {"greeting":"Good morning, John of Seattle. Happy Wednesday!"}
Thu Aug 31 01:07:25 UTC 2017 : Endpoint response headers: {x-amzn-Remapped-Content-
Length=0, x-amzn-RequestId=c0475a28-8de8-11e7-8d3f-4183da788f0f, Connection=keep-
alive, Content-Length=62, Date=Thu, 31 Aug 2017 01:07:25 GMT, X-Amzn-Trace-
Id=root=1-59a7614d-373151b01b0713127e646635;sampled=0, Content-Type=application/json}
Thu Aug 31 01:07:25 UTC 2017 : Method response body after transformations:
 {"greeting":"Good morning, John of Seattle. Happy Wednesday!"}
Thu Aug 31 01:07:25 UTC 2017 : Method response headers: {X-Amzn-Trace-
Id=sampled=0;root=1-59a7614d-373151b01b0713127e646635, Content-Type=application/json}
Thu Aug 31 01:07:25 UTC 2017 : Successfully completed execution
Thu Aug 31 01:07:25 UTC 2017 : Method completed with status: 200
```

The logs show the incoming request before the mapping and the integration request after the mapping. When a test fails, the logs are useful for evaluating whether the original input is correct or the mapping template works correctly.

Deploy the API

The test invocation is a simulation and has limitations. For example, it bypasses any authorization mechanism enacted on the API. To test the API execution in real time, you must deploy the API first. To deploy an API, you create a stage to create a snapshot of the API at that time. The stage name also defines the base path after the API's default host name. The API's root resource is appended after the stage name. When you modify the API, you must redeploy it to a new or existing stage before the changes take effect.

To deploy the API to a stage

1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`test`**.

   > **Note**
   >
   > The input must be UTF-8 encoded (i.e., unlocalized) text.

4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

Under **Stage details**, choose the copy icon to copy your API's invoke URL. The general pattern of the API's base URL is `https://`*`api-id`*`.`*`region`*`.amazonaws.com/`*`stageName`*. For example, the base URL of the API (`beags1mnid`) created in the `us-west-2` region and deployed to the `test` stage is `https://beags1mnid.execute-api.us-west-2.amazonaws.com/test`.

Test the API in a deployment stage

There are several ways you can test a deployed API. For GET requests using only URL path variables or query string parameters, you can enter the API resource URL in a browser. For other methods, you must use more advanced REST API testing utilities, such as POSTMAN or cURL.

To test the API using cURL

1. Open a terminal window on your local computer connected to the internet.
2. To test `POST /Seattle?time=evening`:

   Copy the following cURL command and paste it into the terminal window.

```
curl -v -X POST \
  'https://beags1mnid.execute-api.us-west-2.amazonaws.com/test/Seattle?
time=evening' \
  -H 'content-type: application/json' \
  -H 'day: Thursday' \
  -H 'x-amz-docs-region: us-west-2' \
  -d '{
 "callerName": "John"
}'
```

You should get a successful response with the following payload:

```
{"greeting":"Good evening, John of Seattle. Happy Thursday!"}
```

If you change POST to PUT in this method request, you get the same response.

Clean up

If you no longer need the Lambda functions you created for this walkthrough, you can delete them now. You can also delete the accompanying IAM resources.

> ⚠ **Warning**
>
> If you plan to complete the other walkthroughs in this series, do not delete the Lambda execution role or the Lambda invocation role. If you delete a Lambda function that your APIs rely on, those APIs will no longer work. Deleting a Lambda function cannot be undone. If you want to use the Lambda function again, you must re-create the function.
> If you delete an IAM resource that a Lambda function relies on, that Lambda function will no longer work, and any APIs that rely on that function will no longer work. Deleting an IAM resource cannot be undone. If you want to use the IAM resource again, you must re-create the resource.

To delete the Lambda functions

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. From the list of functions, choose **GetHelloWorld**, choose **Actions**, and then choose **Delete function**. When prompted, choose **Delete** again.
3. From the list of functions, choose **GetHelloWithName**, choose **Actions**, and then choose **Delete function**. When prompted, choose **Delete** again.

To delete the associated IAM resources

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. From **Details**, choose **Roles**.
3. From the list of roles, choose **APIGatewayLambdaExecRole**, choose **Role Actions**, and then choose **Delete Role**. When prompted, choose **Yes, Delete**.
4. From **Details**, choose **Policies**.
5. From the list of policies, choose **APIGatewayLambdaExecPolicy**, choose **Policy Actions**, and then choose **Delete**. When prompted, choose **Delete**.

Tutorial: Create a REST API by importing an example

You can use the Amazon API Gateway console to create and test a simple REST API with the HTTP integration for a PetStore website. The API definition is preconfigured as a OpenAPI 2.0 file. After loading the API definition into API Gateway, you can use the API Gateway console to examine the API's basic structure or simply deploy and test the API.

The PetStore example API supports the following methods for a client to access the HTTP backend website of `http://petstore-demo-endpoint.execute-api.com/petstore/pets`.

> **Note**
>
> This tutorial uses an HTTP endpoint as an example. When you create your own APIs, we recommend you use HTTPS endpoints for your HTTP integrations.

- `GET /`: for read access of the API's root resource that is not integrated with any backend endpoint. API Gateway responds with an overview of the PetStore website. This is an example of the `MOCK` integration type.
- `GET /pets`: for read access to the API's `/pets` resource that is integrated with the like-named backend `/pets` resource. The backend returns a page of available pets in the PetStore. This is an example of the `HTTP` integration type. The URL of the integration endpoint is `http://petstore-demo-endpoint.execute-api.com/petstore/pets`.
- `POST /pets`: for write access to the API's `/pets` resource that is integrated with the backend `/petstore/pets` resource. Upon receiving a correct request, the backend adds the specified pet to the PetStore and returns the result to the caller. The integration is also `HTTP`.

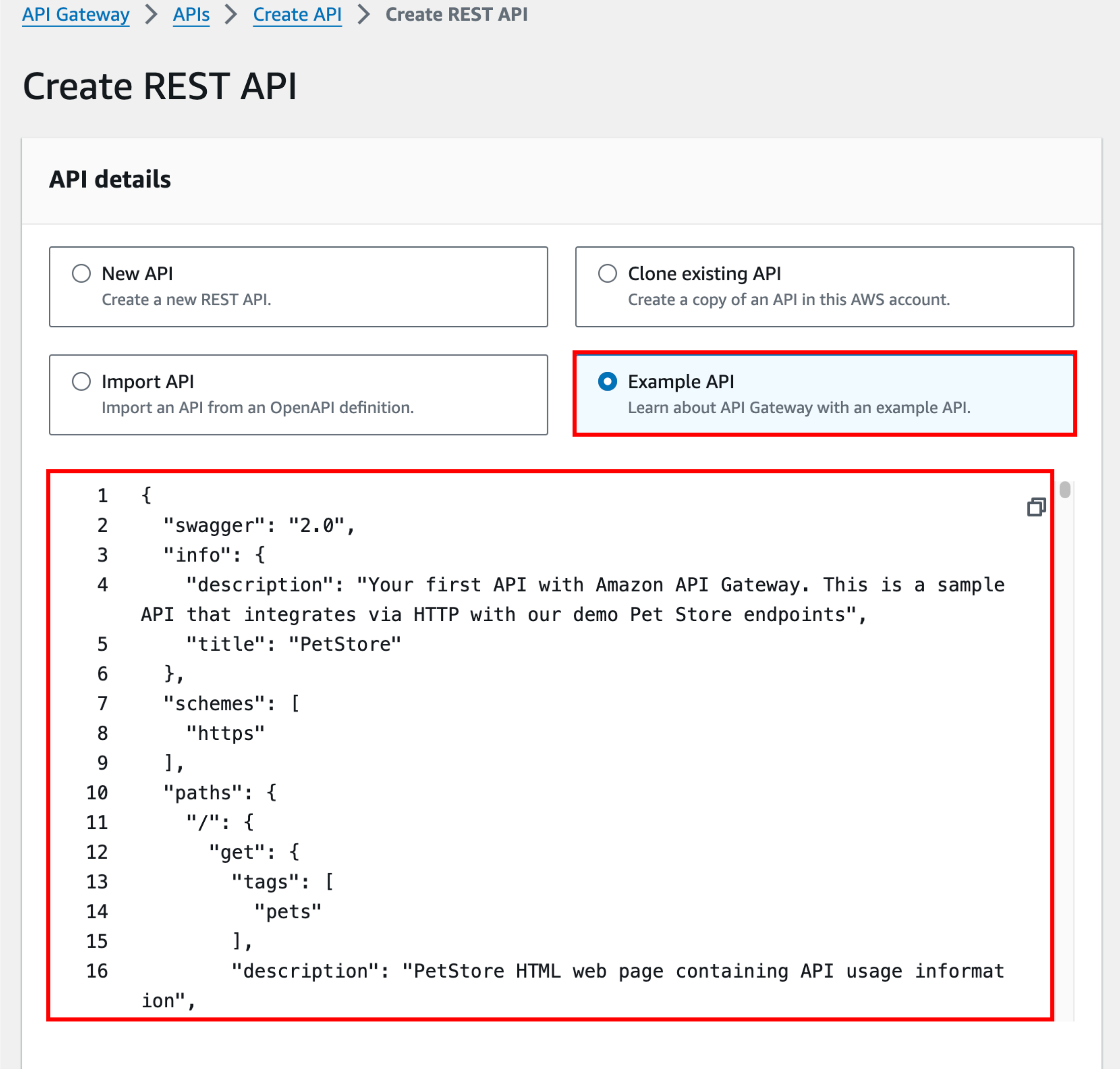
- `GET /pets/{petId}`: for read access to a pet as identified by a `petId` value as specified as a path variable of the incoming request URL. This method also has the `HTTP` integration type. The backend returns the specified pet found in the PetStore. The URL of the backend HTTP endpoint is `http://petstore-demo-endpoint.execute-api.com/petstore/pets/`*n*, where `n` is an integer as the identifier of the queried pet.

The API supports CORS access via the `OPTIONS` methods of the `MOCK` integration type. API Gateway returns the required headers supporting CORS access.

The following procedure walks you through the steps to create and test an API from an example using the API Gateway Console.

To import, build, and test the example API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
   - To create your first API, for **REST API**, choose **Build**.
   - If you've created an API before, choose **Create API**, and then choose **Build** for **REST API**.
3. Under **Create REST API**, choose **Example API** and then choose **Create API** to create the example API.



Create REST API

API details

○ New API
Create a new REST API.

○ Clone existing API
Create a copy of an API in this AWS account.

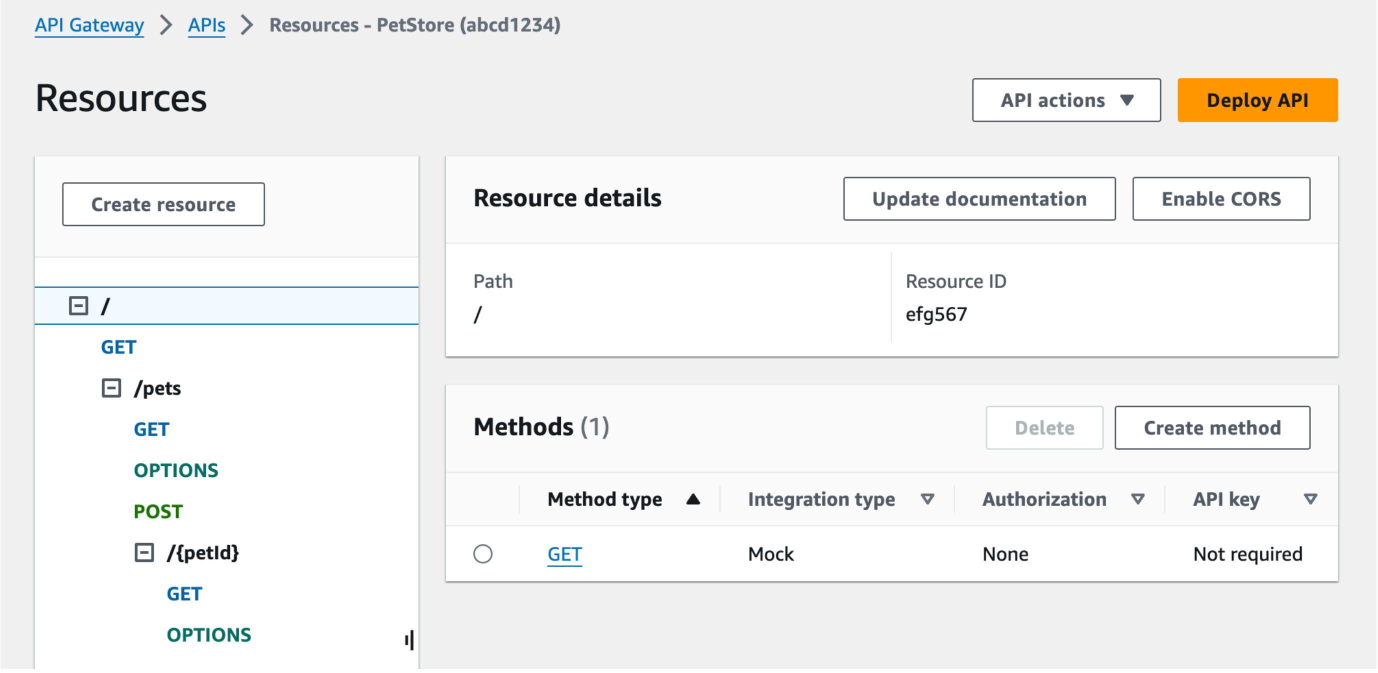
○ Import API
Import an API from an OpenAPI definition.

● Example API
Learn about API Gateway with an example API.

```
{
  "swagger": "2.0",
  "info": {
    "description": "Your first API with Amazon API Gateway. This is a sample API that integrates via HTTP with our demo Pet Store endpoints",
    "title": "PetStore"
  },
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "tags": [
          "pets"
        ],
        "description": "PetStore HTML web page containing API usage information",
```

You can scroll down the OpenAPI definition for details of this example API before choosing **Create API**.

4. In the main navigation pane, choose **Resources**. The newly created API is shown as follows:



⊟ /
GET
⊟ /pets
GET
OPTIONS
POST
⊟ /{petId}
GET
OPTIONS

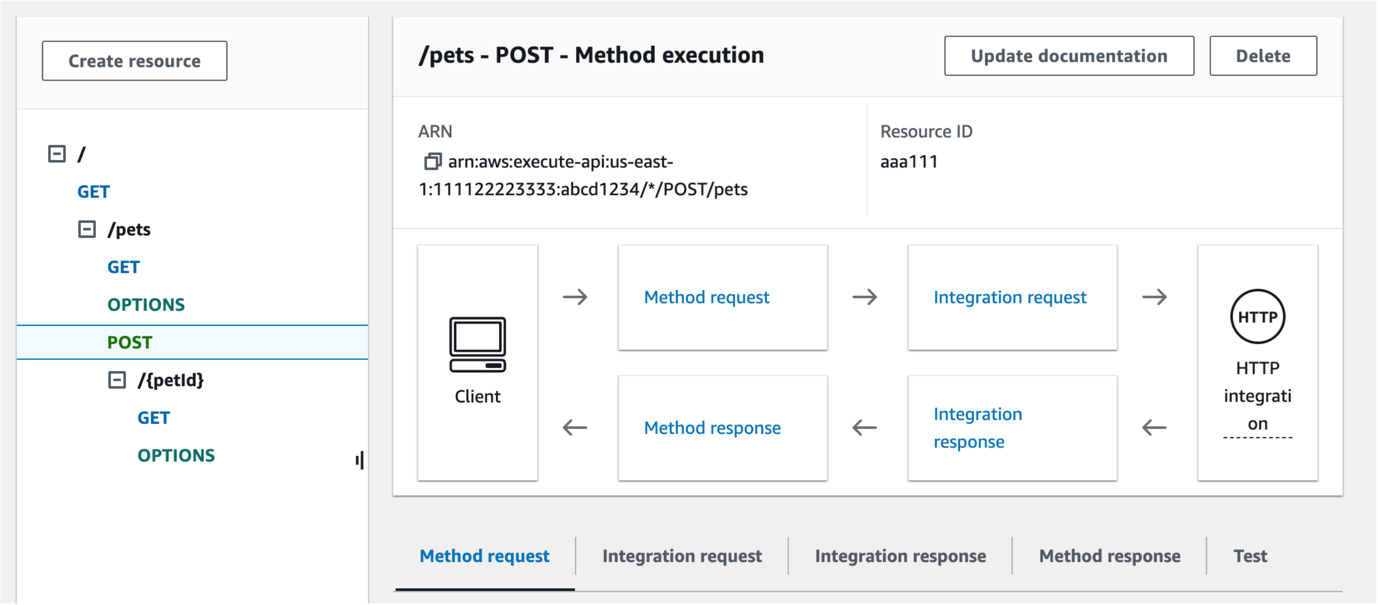
Path: /

Resource ID: efg567

	Method type ▲	Integration type ▽	Authorization ▽	API key ▽
○	GET	Mock	None	Not required

The **Resources** pane shows the structure of the created API as a tree of nodes. API methods defined on each resource are edges of the tree. When a resource is selected, all of its methods are listed in the **Methods** table on the right. Displayed with each method is the method type, integration type, authorization type, and API key requirement.

5. To view the details of a method, to modify its set-up, or to test the method invocation, choose the method name from either the method list or the resource tree. Here, we choose the `POST /pets` method as an illustration:



⊟ /
GET
⊟ /pets
GET
OPTIONS
POST
⊟ /{petId}
GET
OPTIONS

ARN: arn:aws:execute-api:us-east-1:111122223333:abcd1234/*/POST/pets

Resource ID: aaa111

The resulting **Method execution** pane presents a logical view of the chosen (`POST /pets`) method's structure and behaviors.

The **Method request** and **Method response** represent the API's interface with the frontend, and the **Integration request** and **Integration response** represent the API's interface with the backend.

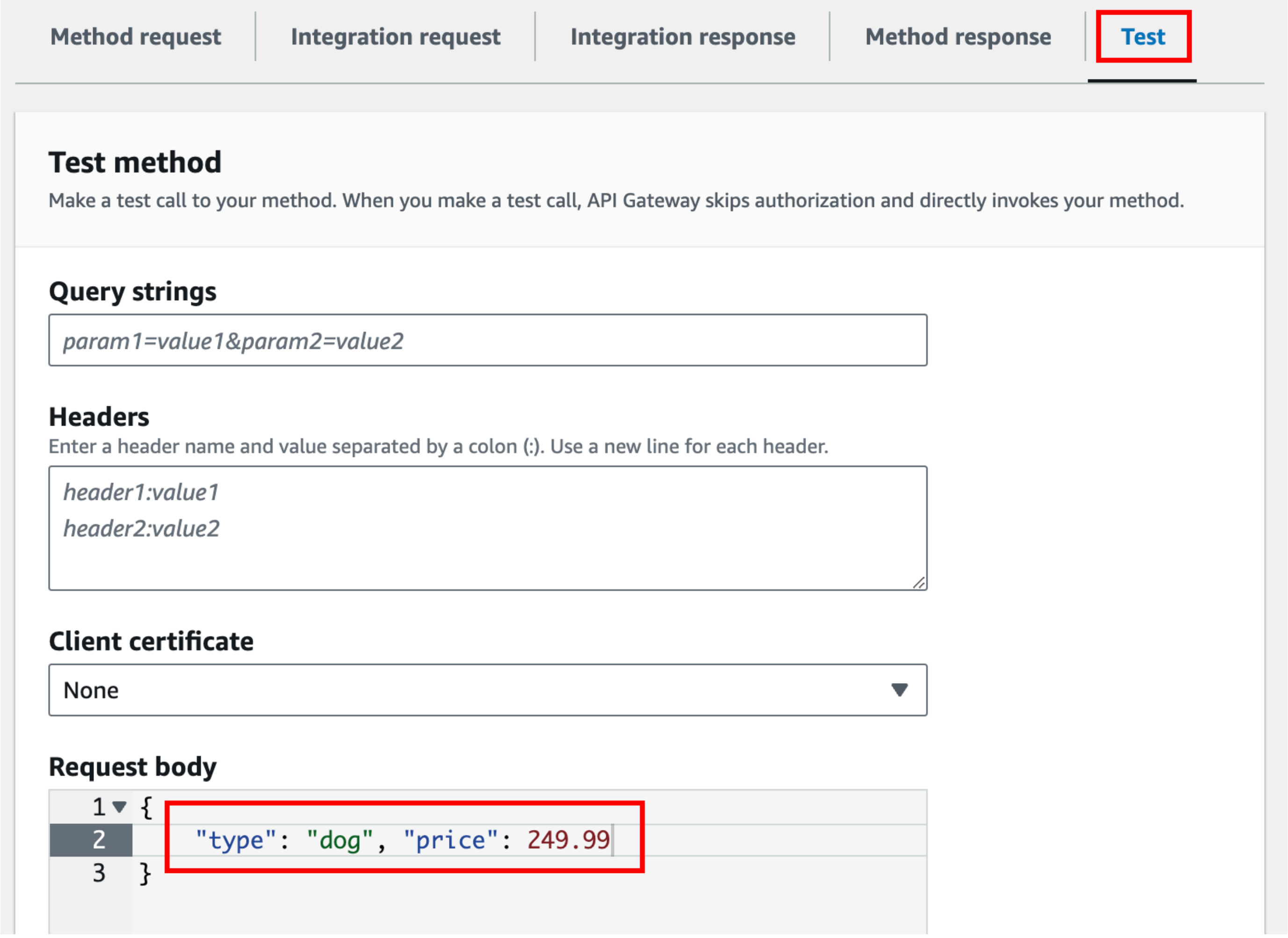
A client uses the API to access a backend feature through the **Method request**. API Gateway translates the client request, if necessary, into the form acceptable to the backend in **Integration request** before forwarding the incoming request to the backend. The transformed request is known as the integration request. Similarly, the backend returns the response to API Gateway in **Integration response**. API Gateway then routes it to **Method Response** before sending it to the client. Again, if necessary, API Gateway can map the backend response data to a form expected by the client.

For the `POST` method on an API resource, the method request payload can be passed through to the integration request without modification, if the method request's payload is of the same format as the integration request's payload.

The `GET /` method request uses the `MOCK` integration type and is not tied to any real backend endpoint. The corresponding **Integration response** is set up to return a static HTML page. When the method is called, the API Gateway simply accepts the request and immediately returns the configured integration response to the client by way of **Method response**. You can use the mock integration to test an API without requiring a backend endpoint. You can also use it to serve a local response, generated from a response body-mapping template.

As an API developer, you control the behaviors of your API's frontend interactions by configuring the method request and a method response. You control the behaviors of your API's backend interactions by setting up the integration request and integration response. These involve data mappings between a method and its corresponding integration. For now, we focus on testing the API to provide an end-to-end user experience.

6. Select the **Test** tab. You might need to choose the right arrow button to show the tab.
7. For example, to test the `POST /pets` method, enter the following **`{"type": "dog","price": 249.99}`** payload into the **Request body**, and then choose **Test**.



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

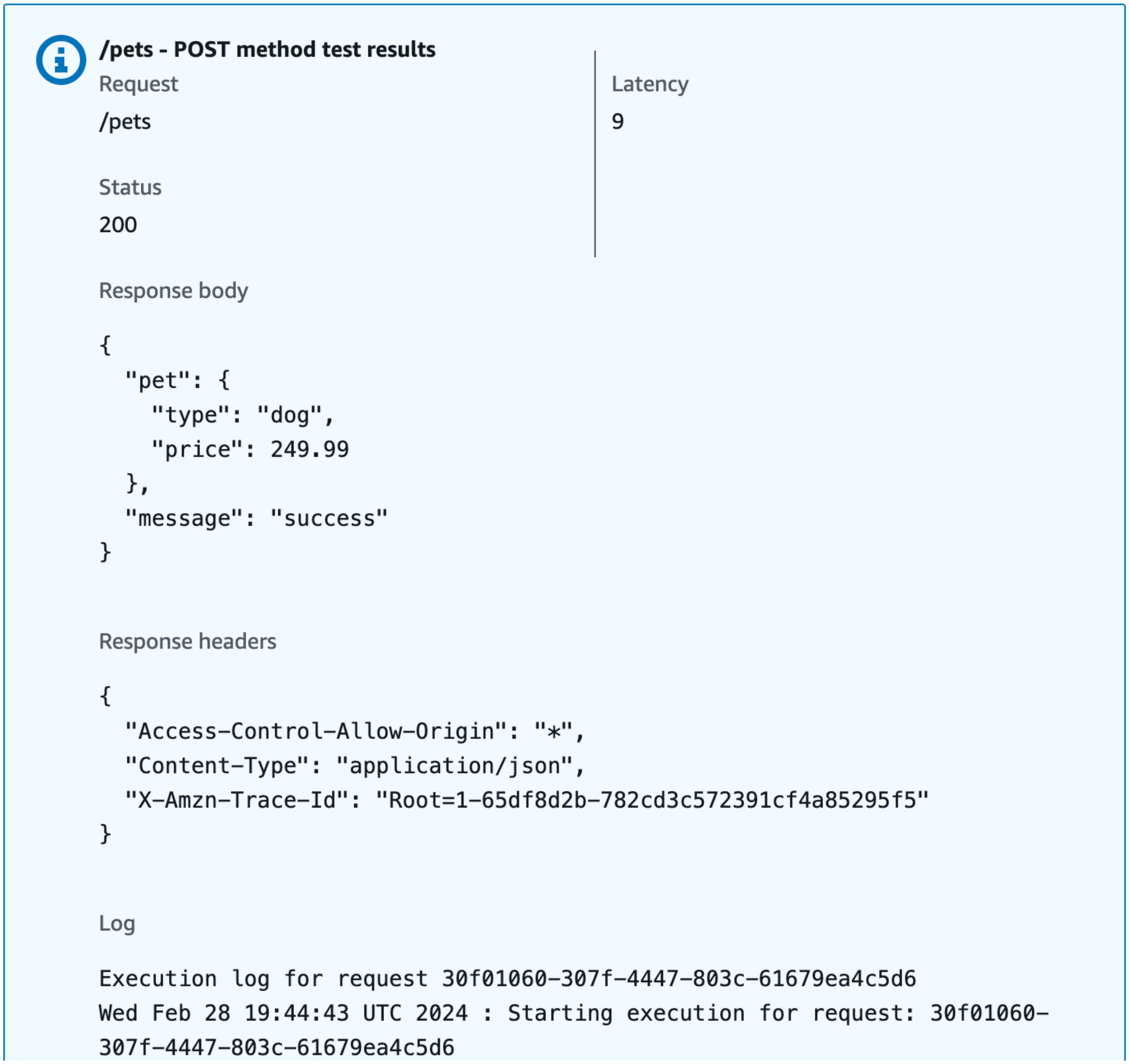
None

Request body

```
{
    "type": "dog", "price": 249.99
}
```

The input specifies the attributes of the pet that we want to add to the list of pets on the PetStore website.

8. The results display as follows:



/pets - POST method test results

Request

/pets

Status

200

Latency

9

Response body

```
{
  "pet": {
    "type": "dog",
    "price": 249.99
  },
  "message": "success"
}
```

Response headers

```
{
  "Access-Control-Allow-Origin": "*",
  "Content-Type": "application/json",
  "X-Amzn-Trace-Id": "Root=1-65df8d2b-782cd3c572391cf4a85295f5"
}
```

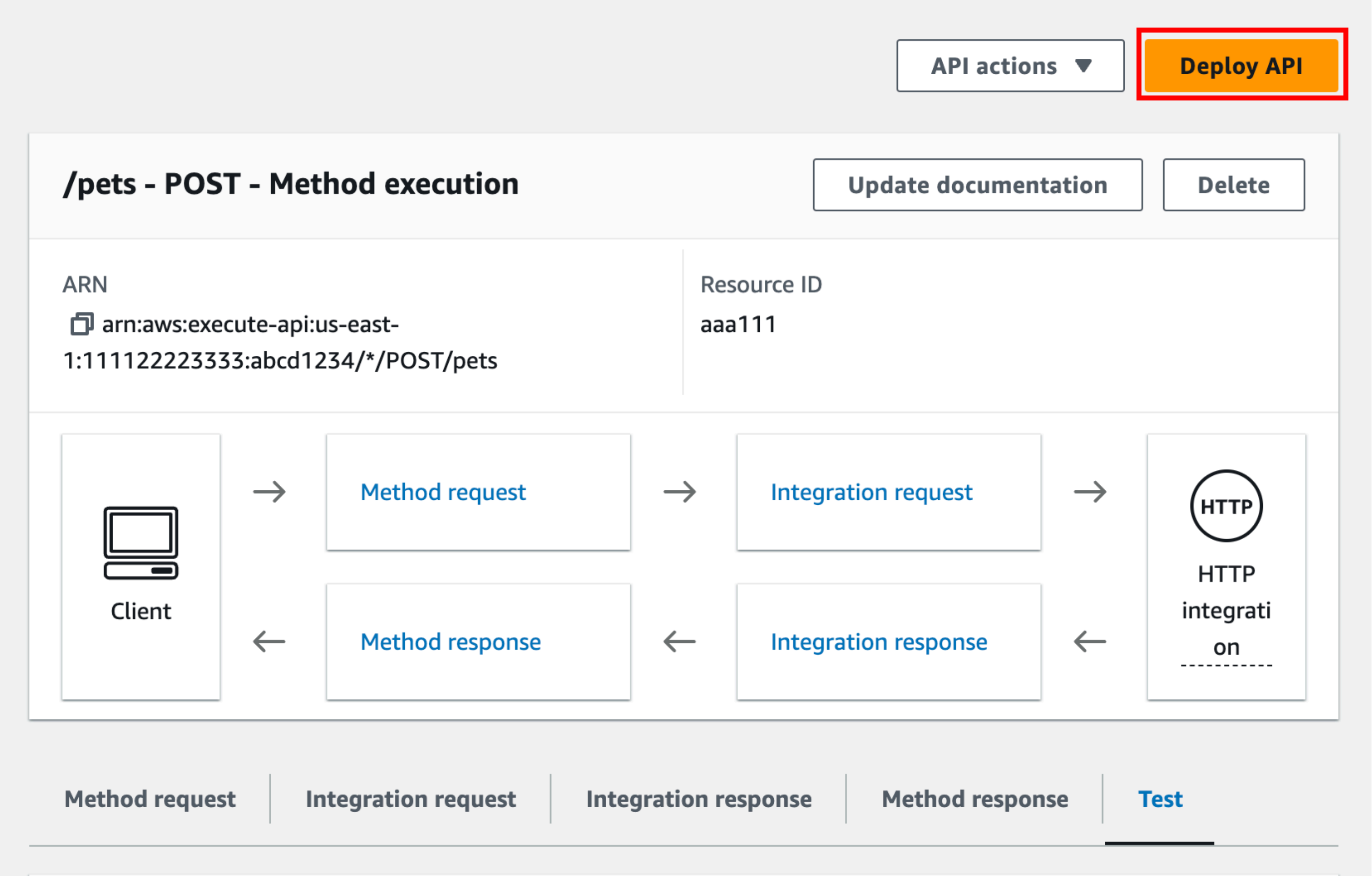
Log

```
Execution log for request 30f01060-307f-4447-803c-61679ea4c5d6
Wed Feb 28 19:44:43 UTC 2024 : Starting execution for request: 30f01060-
307f-4447-803c-61679ea4c5d6
```

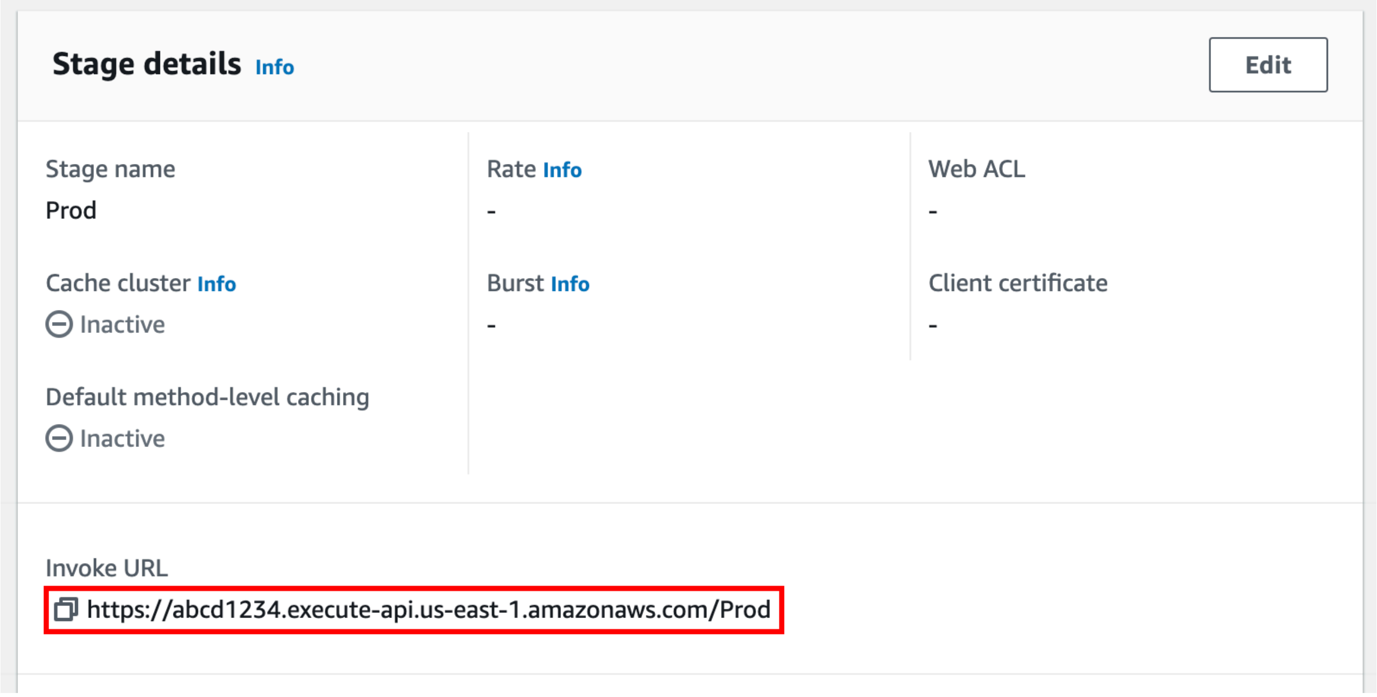
The **Log** entry of the output shows the state changes from the method request to the integration request, and from the integration response to the method response. This can be useful for troubleshooting any mapping errors that cause the request to fail. In this example, no mapping is applied: the method request payload is passed through the integration request to the backend and, similarly, the backend response is passed through the integration response to the method response.

To test the API using a client other than the API Gateway test-invoke-request feature, you must first deploy the API to a stage.

9. To deploy the sample API, choose **Deploy API**.



10. For **Stage**, select **New stage**, and then enter **`test`**.
11. (Optional) For **Description**, enter a description.
12. Choose **Deploy**.
13. In the resulting **Stages** pane, under **Stage details**, the **Invoke URL** displays the URL to invoke the API's GET / method request.



Stage name
Prod

Cache cluster Info
Inactive

Default method-level caching
Inactive

Rate Info
-

Burst Info
-

Web ACL
-

Client certificate
-

Invoke URL
https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod

14. Choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in a web browser. A successful response return the result, generated from the mapping template in the integration response.

15. In the **Stages** navigation pane, expand the **test** stage, select **GET** on `/pets/{petId}`, and then copy the **Invoke URL** value of `https://api-id.execute-api.region.amazonaws.com/test/pets/{petId}`. `{petId}` stands for a path variable.

    Paste the **Invoke URL** value (obtained in the previous step) into the address bar of a browser, replacing `{petId}` by, for example, 1, and press Enter to submit the request. A 200 OK response should return with the following JSON payload:

```
{
  "id": 1,
  "type": "dog",
  "price": 249.99
}
```

    Invoking the API method as shown is possible because its **Authorization** type is set to `NONE`. If the `AWS_IAM` authorization were used, you would sign the request using the Signature Version 4 (SigV4) protocols. For an example of such a request, see the section called "Tutorial: Build an API with HTTP non-proxy integration".

Build an API Gateway REST API with HTTP integration

To build an API with HTTP integration, you can use either the HTTP proxy integration or the HTTP custom integration. We recommend that you use the HTTP proxy integration, whenever possible, for the streamlined API set up while providing versatile and powerful features. The HTTP custom integration can be compelling if it is necessary to transform client request data for the backend or transform the backend response data for the client.

Topics

Tutorial: Build a REST API with HTTP proxy integration

HTTP proxy integration is a simple, powerful, and versatile mechanism to build an API that allows a web application to access multiple resources or features of the integrated HTTP endpoint, for example the entire website, with a streamlined setup of a single API method. In HTTP proxy integration, API Gateway passes the client-submitted method request to the backend. The request data that is passed through includes the request headers, query string parameters, URL path variables, and payload. The backend HTTP endpoint or the web server parses the incoming request data to determine the response that it returns. HTTP proxy integration makes the client and backend interact directly with no intervention from API Gateway after the API method is set up, except for known issues such as unsupported characters, which are listed in the section called "Important notes".

With the all-encompassing proxy resource `{proxy+}`, and the catch-all `ANY` verb for the HTTP method, you can use an HTTP proxy integration to create an API of a single API method. The method exposes the entire set of the publicly accessible HTTP resources and operations of a website. When the backend web server opens more resources for public access, the client can use these new resources with the same API setup. To enable this, the website developer must communicate clearly to the client developer what the new resources are and what operations are applicable for each of them.

As a quick introduction, the following tutorial demonstrates the HTTP proxy integration. In the tutorial, we create an API using the API Gateway console to integrate with the PetStore website through a generic proxy resource `{proxy+}`, and create the HTTP method placeholder of `ANY`.

Topics

- Create an API with HTTP proxy integration using the API Gateway console
- Test an API with HTTP proxy integration

Create an API with HTTP proxy integration using the API Gateway console

The following procedure walks you through the steps to create and test an API with a proxy resource for an HTTP backend using the API Gateway console. The HTTP backend is the `PetStore` website (`http://petstore-demo-endpoint.execute-api.com/petstore/pets`) from Tutorial: Build a REST API with HTTP non-proxy integration, in which screenshots are used as visual aids to illustrate the API Gateway UI elements. If you are new to using the API Gateway console to create an API, you may want to follow that section first.

To create an API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

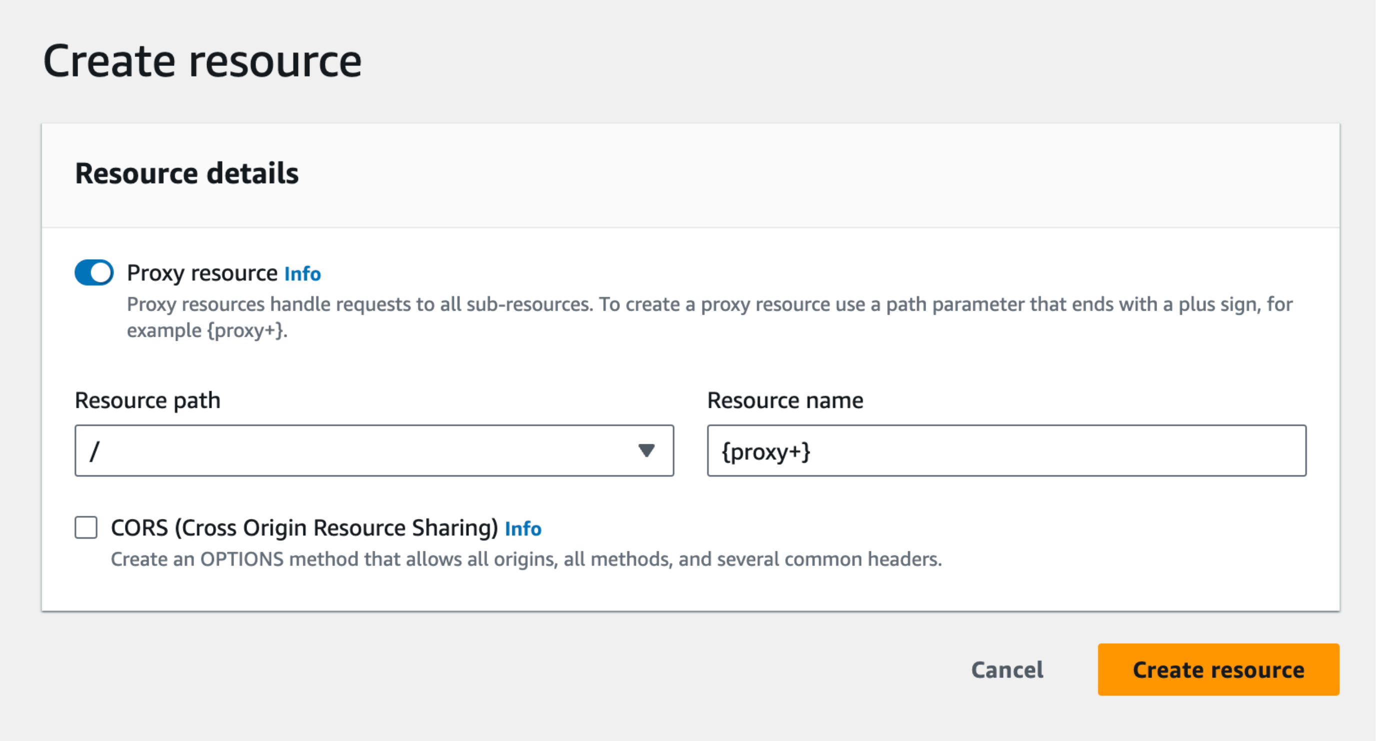
   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. For **API name**, enter **`HTTPProxyAPI`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

In this step, you create a proxy resource path of `{proxy+}`. This is the placeholder of any of the backend endpoints under `http://petstore-demo-endpoint.execute-api.com/`. For example, it can be `petstore`, `petstore/pets`, and `petstore/pets/{petId}`. API Gateway creates the `ANY` method when you create the `{proxy+}` resource and serves as a placeholder for any of the supported HTTP verbs at run time.

To create a /{proxy+} resource

1. Choose your API.

2. In the main navigation pane, choose **Resources**.
3. Choose **Create resource**.
4. Turn on **Proxy resource**.
5. Keep **Resource path** as `/`.
6. For **Resource name**, enter **`{proxy+}`**.
7. Keep **CORS (Cross Origin Resource Sharing)** turned off.
8. Choose **Create resource**.



Create resource

Resource details

Proxy resource Info

Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.

Resource path: /

Resource name: {proxy+}

□ CORS (Cross Origin Resource Sharing) Info

Create an OPTIONS method that allows all origins, all methods, and several common headers.

In this step, you integrate the ANY method with a backend HTTP endpoint, using a proxy integration. In a proxy integration, API Gateway passes the client-submitted method request to the backend with no intervention from API Gateway.

To create an ANY method

1. Choose the **/{proxy+}** resource.
2. Choose the **ANY** method.
3. Under the warning symbol, choose **Edit integration**. You cannot deploy an API that has a method without an integration.
4. For **Integration type**, select **HTTP**.

5. Turn on **HTTP proxy integration**.
6. For **HTTP method**, select **ANY**.
7. For **Endpoint URL**, enter **`http://petstore-demo-endpoint.execute-api.com/{proxy}`**.
8. Choose **Save**.

Test an API with HTTP proxy integration

Whether a particular client request succeeds depends on the following:

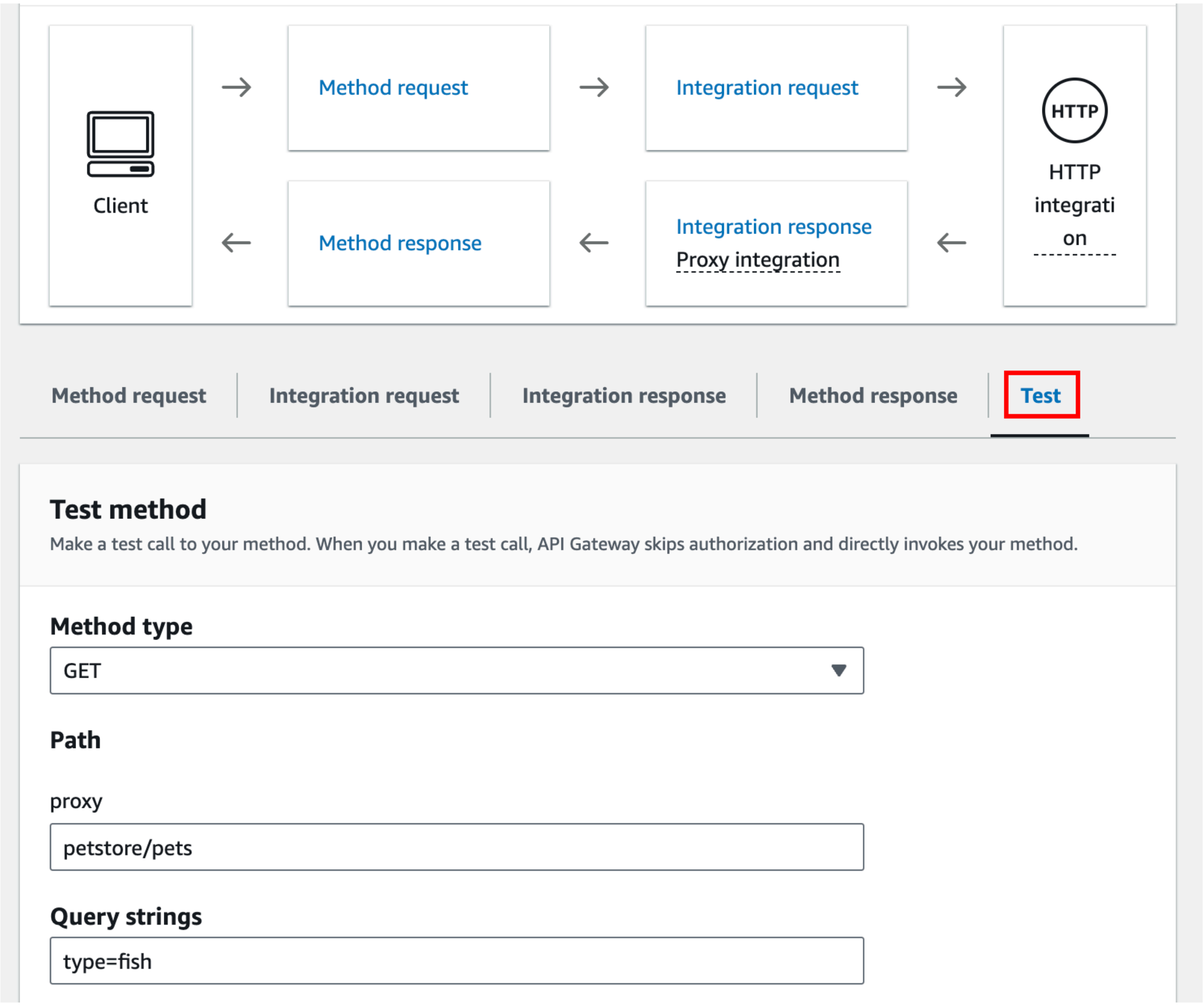
- If the backend has made the corresponding backend endpoint available and, if so, has granted the required access permissions.
- If the client supplies the correct input.

For example, the PetStore API used here does not expose the `/petstore` resource. As such, you get a `404 Resource Not Found` response containing the error message of `Cannot GET /petstore`.

In addition, the client must be able to handle the output format of the backend in order to parse the result correctly. API Gateway does not mediate to facilitate interactions between the client and backend.

To test an API integrated with the PetStore website using HTTP proxy integration through the proxy resource

1. Select the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Method type**, select `GET`.
3. For **Path**, under **proxy**, enter **`petstore/pets`**.
4. For **Query strings**, enter **`type=fish`**.
5. Choose **Test**.



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Method type

GET

Path

proxy

petstore/pets

Query strings

type=fish

Because the backend website supports the `GET /petstore/pets?type=fish` request, it returns a successful response similar to the following:

```
[
  {
    "id": 1,
    "type": "fish",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "fish",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

If you try to call `GET /petstore`, you get a `404` response with an error message of `Cannot GET /petstore`. This is because the backend does not support the specified operation. If you call `GET /petstore/pets/1`, you get a `200 OK` response with the following payload, because the request is supported by the PetStore website.

```
{
  "id": 1,
  "type": "dog",
  "price": 249.99
}
```

You can also use a browser to test your API. Deploy your API and associate it to a stage to create your API's Invoke URL.

To deploy your API

1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`test`**.
4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

Now clients can call your API.

To invoke your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. In the main navigation pane, choose **Stage**.

4. Under **Stage details**, choose the copy icon to copy your API's invoke URL.

   Enter your API's invoke URL in a web browser.

   The full URL should look like `https://abcdef123.execute-api.us-east-2.amazonaws.com/test/petstore/pets?type=fish`.

   Your browser sends a GET request to the API.
5. The result should be the same as returned when you use **Test** in the API Gateway console.

Tutorial: Build a REST API with HTTP non-proxy integration

In this tutorial, you create an API from scratch using the Amazon API Gateway console. You can think of the console as an API design studio and use it to scope the API features, to experiment with its behaviors, to build the API, and to deploy your API in stages.

Topics

- Create an API with HTTP custom integration
- (Optional) Map request parameters

Create an API with HTTP custom integration

This section walks you through the steps to create resources, expose methods on a resource, configure a method to achieve the desired API behaviors, and to test and deploy the API.

In this step, you create an empty API. In the following steps you create resources and methods to connect your API to the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` endpoint, using a non-proxy HTTP integration.

To create an API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.

3. For **API name**, enter **`HTTPNonProxyAPI`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

The **Resources** tree shows the root resource (/) without any methods. In this exercise, we will build the API with the HTTP custom integration of the PetStore website (http://petstore-demo-endpoint.execute-api.com/petstore/pets.) For illustration purposes, we will create a `/pets` resource as a child of the root and expose a GET method on this resource for a client to retrieve a list of available Pets items from the PetStore website.

To create a /pets resource

1. Select the **/** resource, and then choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. Keep **Resource path** as `/`.
4. For **Resource name**, enter **`pets`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

In this step, you create a `GET` method on the **/pets** resource. The `GET` method is integrated with the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` website. Other options for an API method include the following:

- **POST**, primarily used to create child resources.
- **PUT**, primarily used to update existing resources (and, although not recommended, can be used to create child resources).
- **DELETE**, used to delete resources.
- **PATCH**, used to update resources.
- **HEAD**, primarily used in testing scenarios. It is the same as GET but does not return the resource representation.
- **OPTIONS**, which can be used by callers to get information about available communication options for the target service.

For the integration request's **HTTP method**, you must choose one supported by the backend. For `HTTP` or `Mock integration`, it makes sense that the method request and the integration request use the same HTTP verb. For other integration types the method request will likely use an HTTP verb different from the integration request. For example, to call a Lambda function, the integration request must use `POST` to invoke the function, whereas the method request may use any HTTP verb depending on the logic of the Lambda function.

To create a GET method on the /pets resource

1. Select the **/pets** resource.
2. Choose **Create method**.
3. For **Method type**, select **GET**.
4. For **Integration type**, select **HTTP integration**.
5. Keep **HTTP proxy integration** turned off.
6. For **HTTP method**, select **GET**.
7. For **Endpoint URL**, enter **`http://petstore-demo-endpoint.execute-api.com/petstore/pets`**.

   The PetStore website allows you to retrieve a list of `Pet` items by the pet type, such as "Dog" or "Cat", on a given page.
8. For **Content handling**, select **Passthrough**.
9. Choose **URL query string parameters**.

   The PetStore website uses the `type` and `page` query string parameters to accept an input. You add query string parameters to the method request and map them into corresponding query string parameters of the integration request.
10. To add the query string parameters, do the following:

    a. Choose **Add query string**.

    b. For **Name**, enter **`type`**

    c. Keep **Required** and **Caching** turned off.

    Repeat the previous steps to create an additional query string with the name **page**.
11. Choose **Create method**.

The client can now supply a pet type and a page number as query string parameters when submitting a request. These input parameters must be mapped into the integration's query string parameters to forward the input values to our PetStore website in the backend.

To map input parameters to the Integration request

1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. Choose **URL query string parameters**, and then do the following:

    a. Choose **Add query string parameter**.

    b. For **Name**, enter **`type`**.

    c. For **Mapped from**, enter **`method.request.querystring.type`**

    d. Keep **Caching** turned off.

    e. Choose **Add query string parameter**.

    f. For **Name**, enter **`page`**.

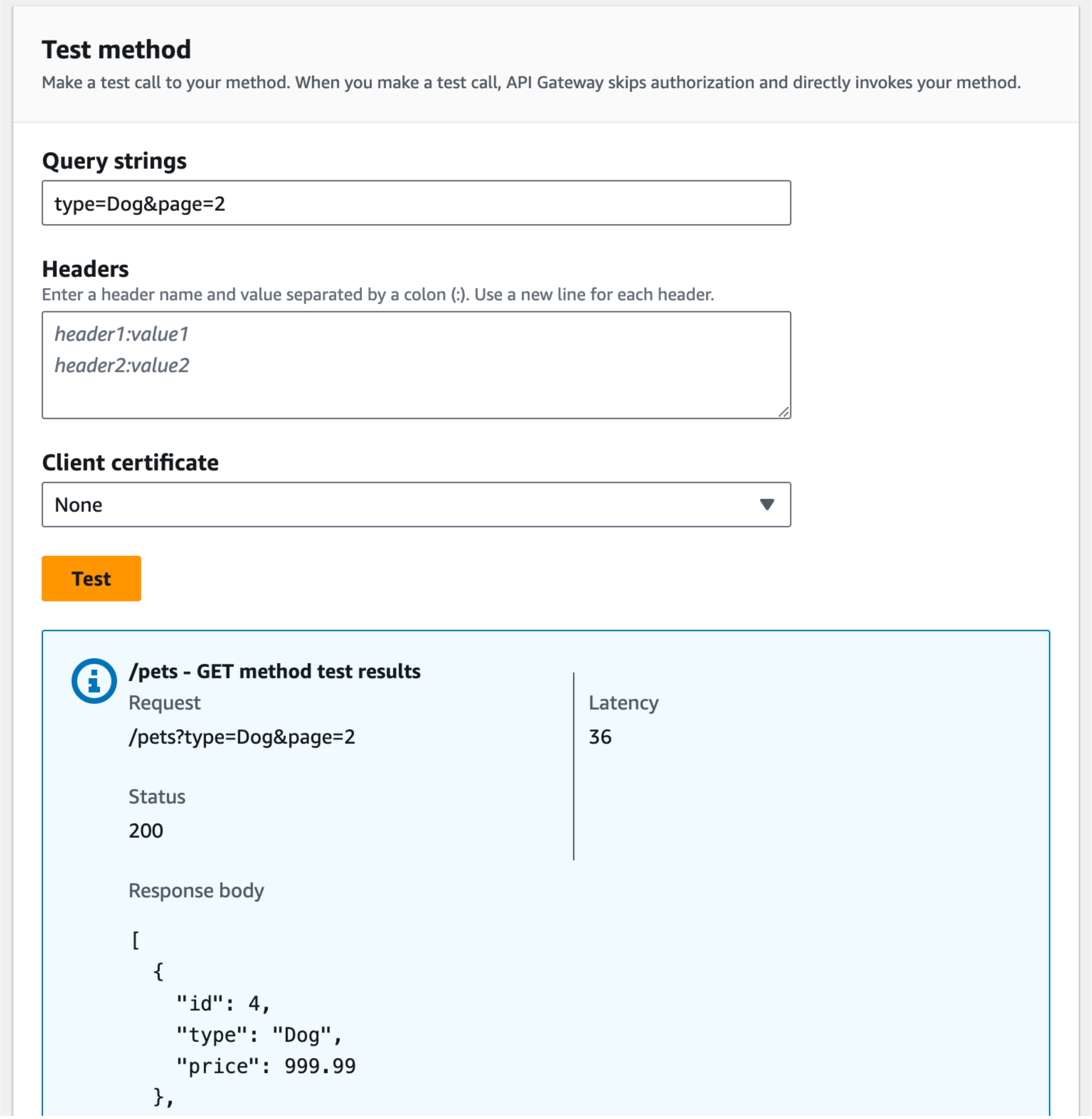
    g. For **Mapped from**, enter **`method.request.querystring.page`**

    h. Keep **Caching** turned off.
3. Choose **Save**.

To test the API

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Query strings**, enter **`type=Dog&page=2`**.
3. Choose **Test**.

    The result is similar to the following:



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

type=Dog&page=2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/pets - GET method test results

Request

/pets?type=Dog&page=2

Status

200

Latency

36

Response body

```
[
  {
    "id": 4,
    "type": "Dog",
    "price": 999.99
  },
```

Now that the test is successful, we can deploy the API to make it publicly available.

4. Choose **Deploy API**.
5. For **Stage**, select **New stage**.
6. For **Stage name**, enter **`Prod`**.
7. (Optional) For **Description**, enter a description.

8. Choose **Deploy**.
9. (Optional) Under **Stage details**, for **Invoke URL**, you can choose the copy icon to copy your API's invoke URL. You can use this with tools such as Postman and cURL to test your API.

If you use an SDK to create a client, you can call the methods exposed by the SDK to sign the request. For implementation details, see the AWS SDK of your choosing.

> **Note**
>
> When changes are made to your API, you must redeploy the API to make the new or updated features available before invoking the request URL again.

(Optional) Map request parameters

Map request parameters for an API Gateway API

This tutorial shows how to create a path parameter of `{petId}` on the API's method request URL to specify an item ID, map it to the `{id}` path parameter in the integration request URL, and send the request to the HTTP endpoint.

> **Note**
>
> If you enter the incorrect case of a letter, such as lowercase letter instead of an uppercase letter, this will cause errors later in the walkthrough.

Step 1: Create resources

In this step, you create a resource with a path parameter {petId}.

To create the {petId} resource

1. Select the **/pets** resource, and then choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. For **Resource path**, select **/pets/**.
4. For **Resource name**, enter **`{petId}`**.

Use the curly braces (`{ }`) around `petId` so that **/pets/{petId}** is displayed.

5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

Step 2: Create and test the methods

In this step, you create a `GET` method with a `{petId}` path parameter.

To set up GET method

1. Select the **/{petId}** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **HTTP integration**.
4. Keep **HTTP proxy integration** turned off.
5. For **HTTP method**, select **GET**.
6. For **Endpoint URL**, enter **`http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`**
7. For **Content handling**, select **Passthrough**.
8. Keep the **Default timeout** turned on.
9. Choose **Create method**.

Now you map the `{petId}` path parameter to the `{id}` path parameter in the HTTP endpoint.

To map the `{petId}` path parameter

1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. Choose **URL path parameters**.
3. API Gateway creates a path parameter for the integration request named **petId**. This doesn't work for your backend. The HTTP endpoint uses `{id}` as the path parameter. Rename **petId** to **`id`**.

   This maps the method request's path parameter of `petId` to the integration request's path parameter of `id`.
4. Choose **Save**.

Now you test the method.

To test the method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. Under **Path** for **petId**, enter **4**.
3. Choose **Test**.

   If successful, **Response body** displays the following:

   ```
   {
     "id": 4,
     "type": "bird",
     "price": 999.99
   }
   ```

Step 3: Deploy the API

In this step, you deploy the API so that you can begin calling it outside of the API Gateway console.

To deploy the API

1. Choose **Deploy API**.
2. For **Stage**, select **Prod**.
3. (Optional) For **Description**, enter a description.
4. Choose **Deploy**.

Step 4: Test the API

In this step, you go outside of the API Gateway console and use your API to access the HTTP endpoint.

1. In the main navigation pane, choose **Stage**.
2. Under **Stage details**, choose the copy icon to copy your API's invoke URL.

   It should look something like this:

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/prod
   ```

3. Enter this URL in the address box of a new browser tab and append `/pets/4` to the URL before you submit your request.
4. The browser will return the following:

```
{
  "id": 4,
  "type": "bird",
  "price": 999.99
}
```

Next steps

You can further customize your API by turning on request validation, transforming data, or creating custom gateway responses.

To explore more ways to customize your API, see the following tutorials:

- For more information about request validation, see Set up basic request validation in API Gateway.
- For information about how to transform request and response payloads, see Set up data transformations in API Gateway.
- For information about how to create custom gateway responses see, Set up a gateway response for a REST API using the API Gateway console.

Tutorial: Build a REST API with API Gateway private integration

You can create an API Gateway API with private integration to provide your customers access to HTTP/HTTPS resources within your Amazon Virtual Private Cloud (Amazon VPC). Such VPC resources are HTTP/HTTPS endpoints on an EC2 instance behind a Network Load Balancer in the VPC. The Network Load Balancer encapsulates the VPC resource and routes incoming requests to the targeted resource.

When a client calls the API, API Gateway connects to the Network Load Balancer through the pre-configured VPC link. A VPC link is encapsulated by an API Gateway resource of VpcLink. It is responsible for forwarding API method requests to the VPC resources and returns backend responses to the caller. For an API developer, a `VpcLink` is functionally equivalent to an integration endpoint.

To create an API with private integration, you must create a new `VpcLink`, or choose an existing one, that is connected to a Network Load Balancer that targets the desired VPC resources. You must have appropriate permissions to create and manage a `VpcLink`. You then set up an API method and integrate it with the `VpcLink` by setting either `HTTP` or `HTTP_PROXY` as the integration type, setting `VPC_LINK` as the integration connection type, and setting the `VpcLink` identifier on the integration `connectionId`.

> **Note**
>
> The Network Load Balancer and API must be owned by the same AWS account.

To quickly get started creating an API to access VPC resources, we walk through the essential steps for building an API with the private integration, using the API Gateway console. Before creating the API, do the following:

1. Create a VPC resource, create or choose a Network Load Balancer under your account in the same region, and add the EC2 instance hosting the resource as a target of the Network Load Balancer. For more information, see Set up a Network Load Balancer for API Gateway private integrations.
2. Grant permissions to create the VPC links for private integrations. For more information, see Grant permissions to create a VPC link.

After creating your VPC resource and your Network Load Balancer with your VPC resource configured in its target groups, follow the instructions below to create an API and integrate it with the VPC resource via a `VpcLink` in a private integration.

To create an API with a private integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. Create an edge-optimized or Regional REST API.

4. Select your API.
5. Choose **Create method**, and then do the following:

    a. For **Method type**, select `GET`.

    b. For **Integration type**, select **VPC link**.

    c. Turn on **VPC proxy integration**.

    d. For **HTTP method**, select `GET`.

    e. For **VPC link**, select **[Use stage variable]** and enter **`${stageVariables.vpcLinkId}`** in the text box below.

        You define the `vpcLinkId` stage variable after deploying the API to a stage and set its value to the ID of the `VpcLink`.

    f. For **Endpoint URL**, enter a URL, for example, `http://myApi.example.com`.

        Here, the host name (for example, `myApi.example.com`) is used to set the `Host` header of the integration request.

    g. Choose **Create method**.

        With the proxy integration, the API is ready for deployment. Otherwise, you need to proceed to set up appropriate method responses and integration responses.

6. Choose **Deploy API**, and then do the following:

    a. For **Stage**, select **New stage**.

    b. For **Stage name**, enter a stage name.

    c. (Optional) For **Description**, enter a description.

    d. Choose **Deploy**.

7. Under the **Stage details** section, note the resulting **Invoke URL**. You need it to invoke the API. Before doing that, you must set up the `vpcLinkId` stage variable.
8. In the **Stages** pane, choose the **Stage variables** tab, and then do the following:

    a. Choose **Manage variables**, and then choose **Add stage variable**.

    b. For **Name**, enter **`vpcLinkId`**.

    c. For **Value**, enter the ID of `VPC_LINK`, for example, *`gix6s7`*.

    d. Choose **Save**.

Using the stage variable, you can easily switch to different VPC links for the API by changing the stage variable value.

Tutorial: Build an API Gateway REST API with AWS integration

Both the Tutorial: Build a Hello World REST API with Lambda proxy integration and Build an API Gateway REST API with Lambda integration topics describe how to create an API Gateway API to expose the integrated Lambda function. In addition, you can create an API Gateway API to expose other AWS services, such as Amazon SNS, Amazon S3, Amazon Kinesis, and even AWS Lambda. This is made possible by the `AWS` integration. The Lambda integration or the Lambda proxy integration is a special case, where the Lambda function invocation is exposed through the API Gateway API.

All AWS services support dedicated APIs to expose their features. However, the application protocols or programming interfaces are likely to differ from service to service. An API Gateway API with the `AWS` integration has the advantage of providing a consistent application protocol for your client to access different AWS services.

In this walkthrough, we create an API to expose Amazon SNS. For more examples of integrating an API with other AWS services, see Amazon API Gateway tutorials and workshops.

Unlike the Lambda proxy integration, there is no corresponding proxy integration for other AWS services. Hence, an API method is integrated with a single AWS action. For more flexibility, similar to the proxy integration, you can set up a Lambda proxy integration. The Lambda function then parses and processes requests for other AWS actions.

API Gateway does not retry when the endpoint times out. The API caller must implement retry logic to handle endpoint timeouts.

This walkthrough builds on the instructions and concepts in Build an API Gateway REST API with Lambda integration. If you have not yet completed that walkthrough, we suggest that you do it first.

Topics

Prerequisites

Before you begin this walkthrough, do the following:

1. Complete the steps in Prerequisites for getting started with API Gateway.
2. Create a new API named `MyDemoAPI`. For more information, see Tutorial: Build a REST API with HTTP non-proxy integration.
3. Deploy the API at least once to a stage named `test`. For more information, see Deploy the API in Build an API Gateway REST API with Lambda integration.
4. Complete the rest of the steps in Build an API Gateway REST API with Lambda integration.
5. Create at least one topic in Amazon Simple Notification Service (Amazon SNS). You will use the deployed API to get a list of topics in Amazon SNS that are associated with your AWS account. To learn how to create a topic in Amazon SNS, see Create a Topic. (You do not need to copy the topic ARN mentioned in step 5.)

Step 1: Create the AWS service proxy execution role

To allow the API to invoke Amazon SNS actions, you must have the appropriate IAM policies attached to an IAM role.

To create the AWS service proxy execution role

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Roles**.
3. Choose **Create role**.
4. Choose **AWS service** under **Select type of trusted entity**, and then select **API Gateway** and select **Allows API Gateway to push logs to CloudWatch Logs**.
5. Choose **Next**, and then choose **Next**.

6. For **Role name**, enter **`APIGatewaySNSProxyPolicy`**, and then choose **Create role**.
7. In the **Roles** list, choose the role you just created. You may need to scroll or use the search bar to find the role.
8. For the selected role, select the **Add permissions** tab.
9. Choose **Attach policies** from the dropdown list.
10. In the search bar, enter **`AmazonSNSReadOnlyAccess`** and choose **Add permissions**.

> **Note**
>
> This tutorial uses a managed policy for simplicity. As a best practice, you should create your own IAM policy to grant the minimum permissions required.

11. Note the newly created **Role ARN**, you will use it later.

Step 2: Create the resource

In this step, you create a resource that enables the AWS service proxy to interact with the AWS service.

To create the resource

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Select the root resource, **/**, represented by a single forward slash (**/**), and then choose **Create resource**.
4. Keep **Proxy resource** turned off.
5. Keep **Resource path** as /.
6. For **Resource name**, enter **`mydemoawsproxy`**.
7. Keep **CORS (Cross Origin Resource Sharing)** turned off.
8. Choose **Create resource**.

Step 3: Create the GET method

In this step, you create a GET method that enables the AWS service proxy to interact with the AWS service.

To create the GET method

1. Select the **/mydemoawsproxy** resource, and then choose **Create method**.
2. For method type, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Amazon SNS topic.
5. For **AWS service**, select **Amazon SNS**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **GET**.
8. For **Action type**, select **Use action name**.
9. For **Action name**, enter **`ListTopics`**.
10. For **Execution role**, enter the role ARN for **`APIGatewaySNSProxyPolicy`**.
11. Choose **Create method**.

Step 4: Specify method settings and test the method

You can now test your GET method to verify that it has been properly set up to list your Amazon SNS topics.

To test the GET method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. Choose **Test**.

   The result displays response similar to the following:

```
{
  "ListTopicsResponse": {
    "ListTopicsResult": {
      "NextToken": null,
      "Topics": [
        {
          "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-1"
        },
        {
          "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-2"
        },
        ...
        {
          "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-N"
        }
      ]
    },
    "ResponseMetadata": {
      "RequestId": "abc1de23-45fa-6789-b0c1-d2e345fa6b78"
    }
  }
}
```

Step 5: Deploy the API

In this step, you deploy the API so that you can call it from outside of the API Gateway console.

To deploy the API

1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`test`**.
4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

Step 6: Test the API

In this step, you go outside of the API Gateway console and use your AWS service proxy to interact with the Amazon SNS service.

1. In the main navigation pane, choose **Stage**.
2. Under **Stage details**, choose the copy icon to copy your API's invoke URL.

   It should look like this:

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/test
   ```
3. Enter the URL into the address box of a new browser tab.
4. Append `/mydemoawsproxy` so that the URL looks like this:

```
https://my-api-id.execute-api.region-id.amazonaws.com/test/mydemoawsproxy
```

Browse to the URL. Information similar to the following should be displayed:

```
{"ListTopicsResponse":{"ListTopicsResult":{"NextToken": null,"Topics":
[{"TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-1"},{"TopicArn":
 "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-2"},...{"TopicArn":
 "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-N}]},"ResponseMetadata":
{"RequestId":"abc1de23-45fa-6789-b0c1-d2e345fa6b78}}}
```

Step 7: Clean up

You can delete the IAM resources the AWS service proxy needs to work.

⚠ **Warning**

If you delete an IAM resource an AWS service proxy relies on, that AWS service proxy and any APIs that rely on it will no longer work. Deleting an IAM resource cannot be undone. If you want to use the IAM resource again, you must re-create it.

To delete the associated IAM resources

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the **Details** area, choose **Roles**.
3. Select **APIGatewayAWSProxyExecRole**, and then choose **Role Actions**, **Delete Role**. When prompted, choose **Yes, Delete**.
4. In the **Details** area, choose **Policies**.
5. Select **APIGatewayAWSProxyExecPolicy**, and then choose **Policy Actions**, **Delete**. When prompted, choose **Delete**.

You have reached the end of this walkthrough. For more detailed discussions about creating API as an AWS service proxy, see Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway, Tutorial: Create a `Calc` REST API with two AWS service integrations and one Lambda non-proxy integration, or Tutorial: Create a REST API as an Amazon Kinesis proxy in API Gateway.

Tutorial: Create a Calc REST API with two AWS service integrations and one Lambda non-proxy integration

The Getting Started non-proxy integration tutorial uses `Lambda Function` integration exclusively. `Lambda Function` integration is a special case of the `AWS Service` integration type that performs much of the integration setup for you, such as automatically adding the required resource-based permissions for invoking the Lambda function. Here, two of the three integrations use `AWS Service` integration. In this integration type, you have more control, but you'll need to manually perform tasks like creating and specifying an IAM role containing appropriate permissions.

In this tutorial, you'll create a `Calc` Lambda function that implements basic arithmetic operations, accepting and returning JSON-formatted input and output. Then you'll create a REST API and integrate it with the Lambda function in the following ways:

1. By exposing a `GET` method on the `/calc` resource to invoke the Lambda function, supplying the input as query string parameters. (`AWS Service` integration)
2. By exposing a `POST` method on the `/calc` resource to invoke the Lambda function, supplying the input in the method request payload. (`AWS Service` integration)
3. By exposing a `GET` on nested `/calc/{operand1}/{operand2}/{operator}` resources to invoke the Lambda function, supplying the input as path parameters. (`Lambda Function` integration)

In addition to trying out this tutorial, you may wish to study the OpenAPI definition file for the `Calc` API, which you can import into API Gateway by following the instructions in the section called "OpenAPI".

Topics

- OpenAPI definitions of sample API integrated with a Lambda function

Create an assumable IAM role

In order for your API to invoke your `Calc` Lambda function, you'll need to have an API Gateway assumable IAM role, which is an IAM role with the following trusted relationship:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

The role you create will need to have Lambda InvokeFunction permission. Otherwise, the API caller will receive a `500 Internal Server Error` response. To give the role this permission, you'll attach the following IAM policy to it:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "*"
        }
    ]
}
```

Here's how to accomplish all this:

Create an API Gateway assumable IAM role

1. Log in to the IAM console.

2. Choose **Roles**.
3. Choose **Create Role**.
4. Under **Select type of trusted entity**, choose **AWS Service**.
5. Under **Choose the service that will use this role**, choose **Lambda**.
6. Choose **Next: Permissions**.
7. Choose **Create Policy**.

   A new **Create Policy** console window will open up. In that window, do the following:

   a. In the **JSON** tab, replace the existing policy with the following:

      ```
      {
          "Version": "2012-10-17",
          "Statement": [
              {
                  "Effect": "Allow",
                  "Action": "lambda:InvokeFunction",
                  "Resource": "*"
              }
          ]
      }
      ```

   b. Choose **Review policy**.

   c. Under **Review Policy**, do the following:

      i. For **Name**, type a name such as **`lambda_execute`**.

      ii. Choose **Create Policy**.

8. In the original **Create Role** console window, do the following:

   a. Under **Attach permissions policies**, choose your **`lambda_execute`** policy from the dropdown list.

      If you don't see your policy in the list, choose the refresh button at the top of the list. (Don't refresh the browser page!)

   b. Choose **Next:Tags**.

   c. Choose **Next:Review**.

   d. For the **Role name**, type a name such as **`lambda_invoke_function_assume_apigw_role`**.

   e. Choose **Create role**.

9. Choose your **`lambda_invoke_function_assume_apigw_role`** from the list of roles.
10. Choose the **Trust relationships** tab.
11. Choose **Edit trust relationship**.
12. Replace the existing policy with the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "lambda.amazonaws.com",
          "apigateway.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

13. Choose **Update Trust Policy**.
14. Make a note of the role ARN for the role you just created. You'll need it later.

Create a `Calc` Lambda function

Next you'll create a Lambda function using the Lambda console.

1. In the Lambda console, choose **Create function**.
2. Choose **Author from Scratch**.
3. For **Name**, enter **`Calc`**.
4. For **Runtime**, choose either the latest supported **Node.js** or **Python** runtime.

5. Choose **Create function**.

6. Copy the following Lambda function in your preferred runtime and paste it into the code editor in the Lambda console.

   Node.js

```
export const handler = async function (event, context) {
  console.log("Received event:", JSON.stringify(event));

  if (
    event.a === undefined ||
    event.b === undefined ||
    event.op === undefined
  ) {
    return "400 Invalid Input";
  }

  const res = {};
  res.a = Number(event.a);
  res.b = Number(event.b);
  res.op = event.op;
  if (isNaN(event.a) || isNaN(event.b)) {
    return "400 Invalid Operand";
  }
  switch (event.op) {
    case "+":
    case "add":
      res.c = res.a + res.b;
      break;
    case "-":
    case "sub":
      res.c = res.a - res.b;
      break;
    case "*":
    case "mul":
      res.c = res.a * res.b;
      break;
    case "/":
    case "div":
      if (res.b == 0) {
        return "400 Divide by Zero";
      } else {
        res.c = res.a / res.b;
      }
      break;
    default:
      return "400 Invalid Operator";
  }

  return res;
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)

    try:
        (event['a']) and (event['b']) and (event['op'])
    except KeyError:
        return '400 Invalid Input'

    try:
        res = {
            "a": float(
                event['a']), "b": float(
                event['b']), "op": event['op']}
    except ValueError:
        return '400 Invalid Operand'

    if event['op'] == '+':
        res['c'] = res['a'] + res['b']
    elif event['op'] == '-':
        res['c'] = res['a'] - res['b']
    elif event['op'] == '*':
        res['c'] = res['a'] * res['b']
    elif event['op'] == '/':
        if res['b'] == 0:
            return '400 Divide by Zero'
        else:
            res['c'] = res['a'] / res['b']
    else:
        return '400 Invalid Operator'
    return res
```

7. Under Execution role, choose **Choose an existing role**.
8. Enter the role ARN for the **`lambda_invoke_function_assume_apigw_role`** role you created earlier.
9. Choose **Deploy**.

This function requires two operands (a and b) and an operator (op) from the `event` input parameter. The input is a JSON object of the following format:

```
{
  "a": "Number" | "String",
  "b": "Number" | "String",
  "op": "String"
}
```

This function returns the calculated result (c) and the input. For an invalid input, the function returns either the null value or the "Invalid op" string as the result. The output is of the following JSON format:

```
{
  "a": "Number",
  "b": "Number",
  "op": "String",
  "c": "Number" | "String"
}
```

You should test the function in the Lambda console before integrating it with the API in the next step.

Test the `Calc` Lambda function

Here's how to test your `Calc` function in the Lambda console:

1. Choose the **Test** tab.

2. For the test event name, enter **`calc2plus5`**.
3. Replace the test event definition with the following:

```
{
  "a": "2",
  "b": "5",
  "op": "+"
}
```

4. Choose **Save**.
5. Choose **Test**.
6. Expand **Execution result: succeeded**. You should see the following:

```
{
  "a": 2,
  "b": 5,
  "op": "+",
  "c": 7
}
```

Create a `Calc` API

The following procedure shows how to create an API for the `Calc` Lambda function you just created. In subsequent sections, you'll add resources and methods to it.

To create an API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. For **API name**, enter **`LambdaCalc`**.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.

6. Choose **Create API**.

Integration 1: Create a GET method with query parameters to call the Lambda function

By creating a GET method that passes query string parameters to the Lambda function, you enable the API to be invoked from a browser. This approach can be useful, especially for APIs that allow open access.

After you create an API, you create a resource. Typically, API resources are organized in a resource tree according to the application logic. For this step, you create a **/calc** resource.

To create a /calc resource

1. Choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. Keep **Resource path** as `/`.
4. For **Resource name**, enter **`calc`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

By creating a GET method that passes query string parameters to the Lambda function, you enable the API to be invoked from a browser. This approach can be useful, especially for APIs that allow open access.

In this method, Lambda requires that the POST request be used to invoke any Lambda function. This example shows that the HTTP method in a frontend method request can be different from the integration request in the backend.

To create a GET method

1. Select the **/calc** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Lambda function.
5. For **AWS service**, select **Lambda**.

6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **POST**.
8. For **Action type**, select **Use path override**. This option allows us to specify the ARN of the Invoke action to execute our `Calc` function.
9. For **Path override**, enter **`2015-03-31/functions/arn:aws:lambda:`*`us-east-2`*`:`*`account-id`*`:function:Calc/invocations`**. For *`account-id`*, enter your AWS account ID. For *`us-east-2`*, enter the AWS Region where you created your Lambda function.
10. For **Execution role**, enter the role ARN for **`lambda_invoke_function_assume_apigw_role`**.
11. Do not change the settings of **Credential cache** and **Default timeout**.
12. Choose **Method request settings**.
13. For **Request validator**, select **Validate query string parameters and headers**.

    This setting will cause an error message to return if the client does not specify the required parameters.
14. Choose **URL query string parameters**.

    Now you set up query string parameters for the **GET** method on the **/calc** resource so it can receive input on behalf of the backend Lambda function.

    To create the query string parameters do the following:

    a. Choose **Add query string**.
    b. For **Name**, enter **`operand1`**.
    c. Turn on **Required**.
    d. Keep **Caching** turned off.

    Repeat the same steps and create a query string named **`operand2`** and a query string named **`operator`**.
15. Choose **Create method**.

Now, you create a mapping template to translate the client-supplied query strings to the integration request payload as required by the `Calc` function. This template maps the three query string parameters declared in **Method request** into designated property values of the JSON object as the input to the backend Lambda function. The transformed JSON object will be included as the integration request payload.

To map input parameters to the integration request

1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. For **Request body passthrough**, select **When there are no templates defined (recommended)**.
3. Choose **Mapping templates**.
4. Choose **Add mapping template**.
5. For **Content type**, enter **`application/json`**.
6. For **Template body**, enter the following code:

```
{
    "a":  "$input.params('operand1')",
    "b":  "$input.params('operand2')",
    "op": "$input.params('operator')"
}
```

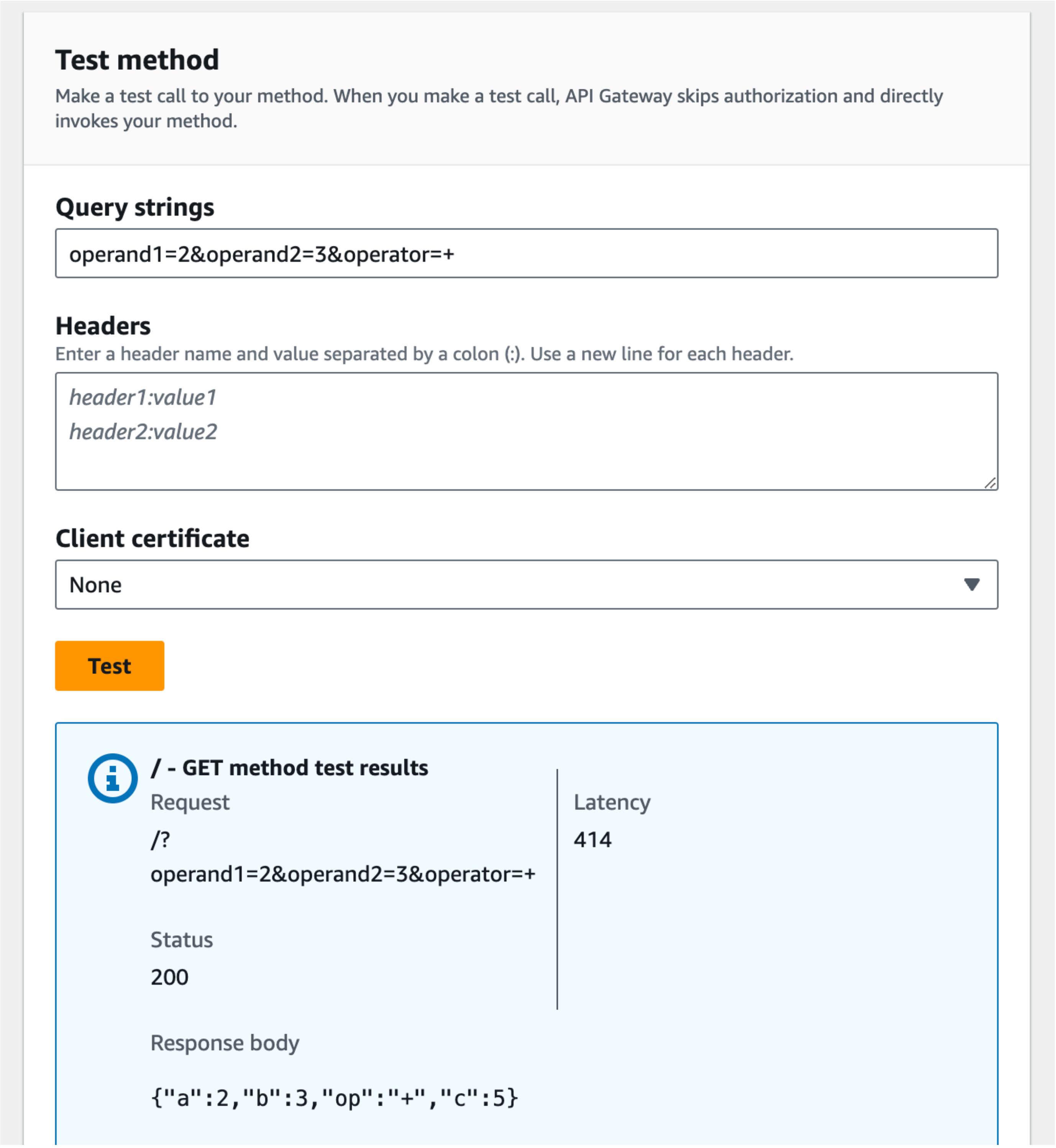
7. Choose **Save**.

You can now test your GET method to verify that it has been properly set up to invoke the Lambda function.

To test the GET method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Query strings**, enter **`operand1=2&operand2=3&operator=+`**.
3. Choose **Test**.

   The results should look similar to this:



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

operand1=2&operand2=3&operator=+

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1

header2:value2

Client certificate

None

/ - GET method test results

Request

/?
operand1=2&operand2=3&operator=+

Latency

414

Status

200

Response body

```
{"a":2,"b":3,"op":"+","c":5}
```

Integration 2: Create a POST method with a JSON payload to call the Lambda function

By creating a POST method with a JSON payload to call the Lambda function, you make it so that the client must provide the necessary input to the backend function in the request body. To ensure that the client uploads the correct input data, you'll enable request validation on the payload.

To create a POST method with a JSON payload

1. Select the **/calc** resource, and then choose **Create method**.
2. For **Method type**, select **POST**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Lambda function.
5. For **AWS service**, select **Lambda**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **POST**.
8. For **Action type**, select **Use path override**. This option allows us to specify the ARN of the Invoke action to execute our `Calc` function.
9. For **Path override**, enter **`2015-03-31/functions/arn:aws:lambda:`*`us-east-2`*`:`*`account-id`*`:function:Calc/invocations`**. For *`account-id`*, enter your AWS account ID. For *`us-east-2`*, enter the AWS Region where you created your Lambda function.
10. For **Execution role**, enter the role ARN for **`lambda_invoke_function_assume_apigw_role`**.
11. Do not change the settings of **Credential cache** and **Default timeout**.
12. Choose **Create method**.

Now you create an **input** model to describe the input data structure and validate the incoming request body.

To create an input model

1. In the main navigation pane, choose **Models**.
2. Choose **Create model**.
3. For **Name**, enter **`input`**.
4. For **Content type**, enter **`application/json`**.

If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **`$default`**.

5. For **Model schema**, enter the following model:

```
{
    "type":"object",
    "properties":{
        "a":{"type":"number"},
        "b":{"type":"number"},
        "op":{"type":"string"}
    },
    "title":"input"
}
```

6. Choose **Create model**.

You now create an **output** model. This model describes the data structure of the calculated output from the backend. It can be used to map the integration response data to a different model. This tutorial relies on the passthrough behavior and does not use this model.

To create an output model

1. Choose **Create model**.
2. For **Name**, enter **`output`**.
3. For **Content type**, enter **`application/json`**.

   If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **`$default`**.

4. For **Model schema**, enter the following model:

```
{
    "type":"object",
    "properties":{
        "c":{"type":"number"}
    },
    "title":"output"
}
```

5. Choose **Create model**.

You now create a **result** model. This model describes the data structure of the returned response data. It references both the **input** and **output** schemas defined in your API.

To create a result model

1. Choose **Create model**.
2. For **Name**, enter **`result`**.
3. For **Content type**, enter **`application/json`**.

   If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **`$default`**.
4. For **Model schema**, enter the following model with your *`restapi-id`*. Your *`restapi-id`* is listed in parenthesis at the top of the console in the following flow: `API Gateway > APIs > LambdaCalc (`*`abc123`*`).`

```
{
    "type":"object",
    "properties":{
        "input":{
            "$ref":"https://apigateway.amazonaws.com/restapis/restapi-id/models/
input"
        },
        "output":{
            "$ref":"https://apigateway.amazonaws.com/restapis/restapi-id/models/
output"
        }
    },
    "title":"result"
}
```

5. Choose **Create model**.

You now configure the method request of your POST method to enable request validation on the incoming request body.

To enable request validation on the POST method

1. In the main navigation pane, choose **Resources**, and then select the `POST` method from the resource tree.
2. On the **Method request** tab, under **Method request settings**, choose **Edit**.

3. For **Request validator**, select **Validate body**.
4. Choose **Request body**, and then choose **Add model**.
5. For **Content type**, enter **`application/json`**.

   If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **`$default`**.
6. For **Model**, select **input**.
7. Choose **Save**.

You can now test your `POST` method to verify that it has been properly set up to invoke the Lambda function.

To test the `POST` method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Request body**, enter the following JSON payload.

```
{
    "a": 1,
    "b": 2,
    "op": "+"
}
```

3. Choose **Test**.

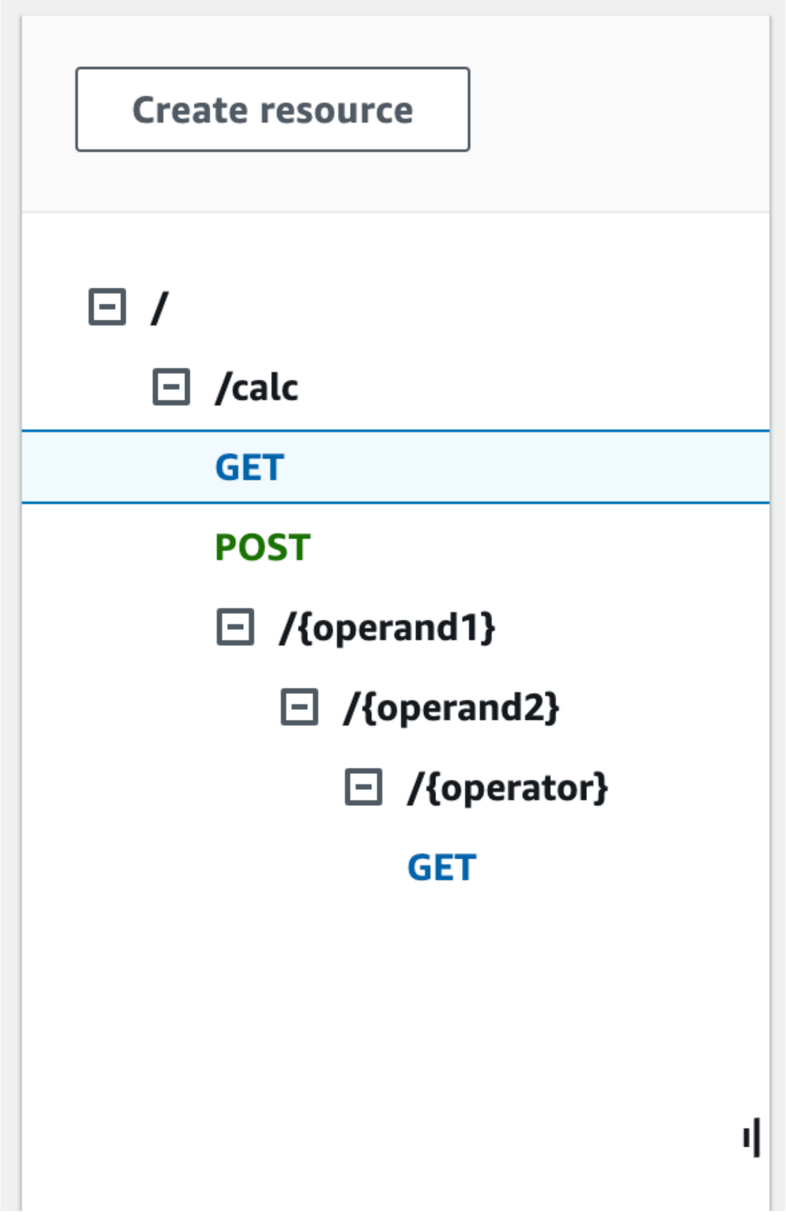
   You should see the following output:

```
{
  "a": 1,
  "b": 2,
  "op": "+",
  "c": 3
}
```

Integration 3: Create a GET method with path parameters to call the Lambda function

Now you'll create a GET method on a resource specified by a sequence of path parameters to call the backend Lambda function. The path parameter values specify the input data to the Lambda function. You'll use a mapping template to map the incoming path parameter values to the required integration request payload.

The resulting API resource structure will look like this:



To create a /{operand1}/{operand2}/{operator} resource

1. Choose **Create resource**.
2. For **Resource path**, select `/calc`.
3. For **Resource name**, enter **`{operand1}`**.
4. Keep **CORS (Cross Origin Resource Sharing)** turned off.
5. Choose **Create resource**.
6. For **Resource path**, select `/calc/{operand1}/`.
7. For **Resource name**, enter **`{operand2}`**.
8. Keep **CORS (Cross Origin Resource Sharing)** turned off.
9. Choose **Create resource**.
10. For **Resource path**, select `/calc/{operand1}/{operand2}/`.
11. For **Resource name**, enter **`{operator}`**.
12. Keep **CORS (Cross Origin Resource Sharing)** turned off.
13. Choose **Create resource**.

This time you'll use the built-in Lambda integration in the API Gateway console to set up the method integration.

To set up a method integration

1. Select the **/{operand1}/{operand2}/{operator}** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **Lambda**.
4. Keep **Lambda proxy integration** turned off.
5. For **Lambda function**, select the AWS Region where you created your Lambda function and enter **`Calc`**.
6. Keep **Default timeout** turned on.
7. Choose **Create method**.

You now create a mapping template to map the three URL path parameters, declared when the **/calc/{operand1}/{operand2}/{operator}** resource was created, into designated property values in the JSON object. Because URL paths must be URL-encoded, the division operator must be specified as %2F instead of /. This template translates the %2F into '/' before passing it to the Lambda function.

To create a mapping template

1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. For **Request body passthrough**, select **When there are no templates defined (recommended)**.
3. Choose **Mapping templates**.
4. For **Content type**, enter **`application/json`**.
5. For **Template body**, enter the following code:

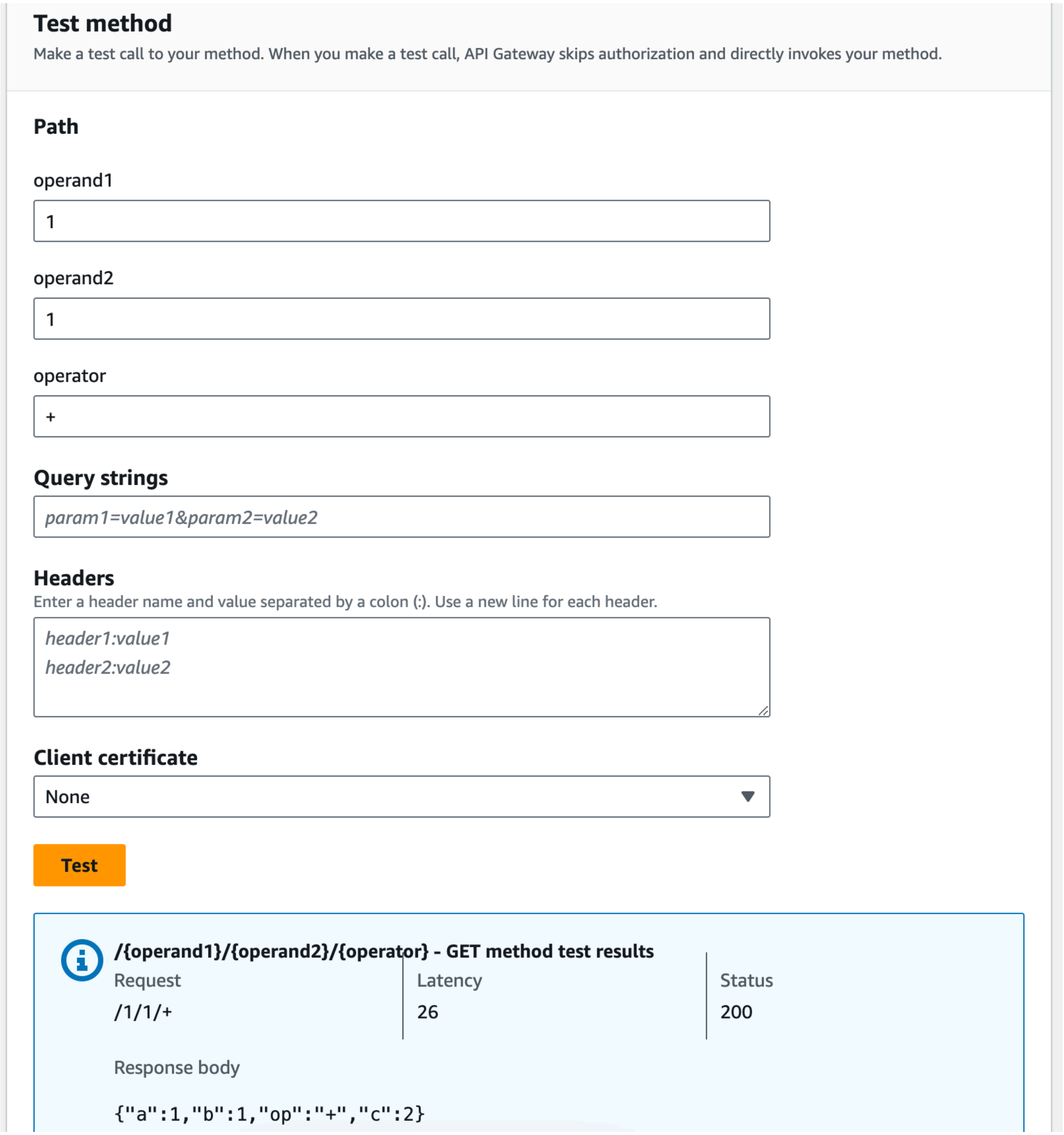
```
{
   "a": "$input.params('operand1')",
   "b": "$input.params('operand2')",
   "op":
 #if($input.params('operator')=='%2F')"/"#{else}"$input.params('operator')"#end
}
```

6. Choose **Save**.

You can now test your GET method to verify that it has been properly set up to invoke the Lambda function and pass the original output through the integration response without mapping.

To test the GET method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For the **Path**, do the following:
   a. For **operand1**, enter **1**.
   b. For **operand2**, enter **1**.
   c. For **operator**, enter **+**.
3. Choose **Test**.
4. The result should look like this:



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

operand1: 1

operand2: 1

operator: +

Query strings: *param1=value1&param2=value2*

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate: None

/{operand1}/{operand2}/{operator} - GET method test results

Request	Latency	Status
/1/1/+	26	200

Response body

```
{"a":1,"b":1,"op":"+","c":2}
```

Next, you model the data structure of the method response payload after the `result` schema.

By default, the method response body is assigned an empty model. This will cause the integration response body to be passed through without mapping. However, when you generate an SDK for one of the strongly-type languages, such as Java or Objective-C, your SDK users will receive an empty object as the result. To ensure that both the REST client and SDK clients receive the desired result, you must model the response data using a predefined schema. Here you'll define a model for the method response body and to construct a mapping template to translate the integration response body into the method response body.

To create a method response

1. On the **Method response** tab, under **Response 200**, choose **Edit**.
2. Under **Response body**, choose **Add model**.
3. For **Content type**, enter **`application/json`**.
4. For **Model**, select **result**.
5. Choose **Save**.

Setting the model for the method response body ensures that the response data will be cast into the `result` object of a given SDK. To make sure that the integration response data is mapped accordingly, you'll need a mapping template.

To create a mapping template

1. On the **Integration response** tab, under **Default - Response**, choose **Edit**.
2. Choose **Mapping templates**.
3. For **Content type**, enter **`application/json`**.
4. For **Generate template**, select **result**.
5. Modify the generated mapping template to match the following:

```
#set($inputRoot = $input.path('$'))
{
  "input" : {
    "a" : $inputRoot.a,
    "b" : $inputRoot.b,
    "op" : "$inputRoot.op"
  },
  "output" : {
    "c" : $inputRoot.c
  }
}
```

6. Choose **Save**.

To test the mapping template

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For the **Path**, do the following:
   a. For **operand1**, enter **1**.
   b. For **operand2**, enter **2**.
   c. For **operator**, enter **+**.
3. Choose **Test**.
4. The result will look like the following:

```
{
  "input": {
    "a": 1,
    "b": 2,
    "op": "+"
  },
  "output": {
    "c": 3
  }
}
```

At this point, you can only call the API using the **Test** feature in the API Gateway console. To make it available to clients, you'll need to deploy your API. Always be sure to redeploy your API whenever you add, modify, or delete a resource or method, update a data mapping, or update stage settings. Otherwise, new features or updates will not be available to clients of your API. as follows:

To deploy the API

1. Choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`Prod`**.
4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.
6. (Optional) Under **Stage details**, for **Invoke URL**, you can choose the copy icon to copy your API's invoke URL. You can use this with tools such as Postman and cURL to test your API.

Note

Always redeploy your API whenever you add, modify, or delete a resource or method, update a data mapping, or update stage settings. Otherwise, new features or updates will not be available to clients of your API.

OpenAPI definitions of sample API integrated with a Lambda function

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2017-04-20T04:08:08Z",
    "title": "LambdaCalc"
  },
  "host": "uojnr9hd57.execute-api.us-east-1.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/calc": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "operand2",
            "in": "query",
            "required": true,
            "type": "string"
          },
          {
            "name": "operator",
            "in": "query",
            "required": true,
            "type": "string"
          },
          {
            "name": "operand1",
            "in": "query",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            },
            "headers": {
              "operand_1": {
                "type": "string"
              },
              "operand_2": {
                "type": "string"
              },
              "operator": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator": "Validate query string parameters
 and headers",
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.operator": "integration.response.body.op",
                "method.response.header.operand_2": "integration.response.body.b",
                "method.response.header.operand_1": "integration.response.body.a"
              },
              "responseTemplates": {
                "application/json": "#set($res = $input.path('$'))\n{\n    \"result
\": \"$res.a, $res.b, $res.op => $res.c\",\n  \"a\" : \"$res.a\",\n  \"b\" :
 \"$res.b\",\n  \"op\" : \"$res.op\",\n  \"c\" : \"$res.c\"\n}"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n    \"a\":  \"$input.params('operand1')\",\n
 \"b\":  \"$input.params('operand2')\", \n    \"op\": \"$input.params('operator')\"
  \n}"
          },
          "type": "aws"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "Input",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Input"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-request-validator": "Validate body",
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n  \"a
\" : $inputRoot.a,\n  \"b\" : $inputRoot.b,\n  \"op\" : $inputRoot.op,\n  \"c\" :
 $inputRoot.c\n}"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "type": "aws"
        }
      }
    },
    "/calc/{operand1}/{operand2}/{operator}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "operand2",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "operator",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "operand1",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n   \"a\": \"$input.params('operand1')\",
\n   \"b\": \"$input.params('operand2')\",\n   \"op\":
 #if($input.params('operator')=='%2F')\"/\"#{else}\"$input.params('operator')\"#end
\n   \n}"
          },
          "contentHandling": "CONVERT_TO_TEXT",
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
    "Input": {
      "type": "object",
      "required": [
        "a",
        "b",
        "op"
      ],
      "properties": {
        "a": {
          "type": "number"
        },
        "b": {
          "type": "number"
        },
        "op": {
          "type": "string",
          "description": "binary op of ['+', 'add', '-', 'sub', '*', 'mul', '%2F',
'div']"
        }
      },
      "title": "Input"
    },
    "Output": {
      "type": "object",
      "properties": {
        "c": {
          "type": "number"
        }
      },
      "title": "Output"
    },
    "Result": {
      "type": "object",
      "properties": {
        "input": {
          "$ref": "#/definitions/Input"
        },
        "output": {
          "$ref": "#/definitions/Output"
        }
      },
      "title": "Result"
    }
  },
  "x-amazon-apigateway-request-validators": {
    "Validate body": {
      "validateRequestParameters": false,
      "validateRequestBody": true
    },
    "Validate query string parameters and headers": {
      "validateRequestParameters": true,
      "validateRequestBody": false
    }
  }
}
```

Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway

As an example to showcase using a REST API in API Gateway to proxy Amazon S3, this section describes how to create and configure a REST API to expose the following Amazon S3 operations:

- Expose GET on the API's root resource to list all of the Amazon S3 buckets of a caller.
- Expose GET on a Folder resource to view a list of all of the objects in an Amazon S3 bucket.
- Expose GET on a Folder/Item resource to view or download an object from an Amazon S3 bucket.

You might want to import the sample API as an Amazon S3 proxy, as shown in OpenAPI definitions of the sample API as an Amazon S3 proxy. This sample contains more exposed methods. For instructions on how to import an API using the OpenAPI definition, see Configuring a REST API using OpenAPI.

> **Note**
>
> To integrate your API Gateway API with Amazon S3, you must choose a region where both the API Gateway and Amazon S3 services are available. For region availability, see Amazon API Gateway Endpoints and Quotas.

Topics

- Set up IAM permissions for the API to invoke Amazon S3 actions
- Create API resources to represent Amazon S3 resources
- Expose an API method to list the caller's Amazon S3 buckets
- Expose API methods to access an Amazon S3 bucket
- Expose API methods to access an Amazon S3 object in a bucket

- OpenAPI definitions of the sample API as an Amazon S3 proxy
- Call the API using a REST API client

Set up IAM permissions for the API to invoke Amazon S3 actions

To allow the API to invoke Amazon S3 actions, you must have the appropriate IAM policies attached to an IAM role.

To create the AWS service proxy execution role

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Roles**.
3. Choose **Create role**.
4. Choose **AWS service** under **Select type of trusted entity**, and then select **API Gateway** and select **Allows API Gateway to push logs to CloudWatch Logs**.
5. Choose **Next**, and then choose **Next**.
6. For **Role name**, enter **`APIGatewayS3ProxyPolicy`**, and then choose **Create role**.
7. In the **Roles** list, choose the role you just created. You may need to scroll or use the search bar to find the role.
8. For the selected role, select the **Add permissions** tab.
9. Choose **Attach policies** from the dropdown list.
10. In the search bar, enter **`AmazonS3FullAccess`** and choose **Add permissions**.

    > **Note**
    >
    > This tutorial uses a managed policy for simplicity. As a best practice, you should create your own IAM policy to grant the minimum permissions required.

11. Note the newly created **Role ARN**, you will use it later.

Create API resources to represent Amazon S3 resources

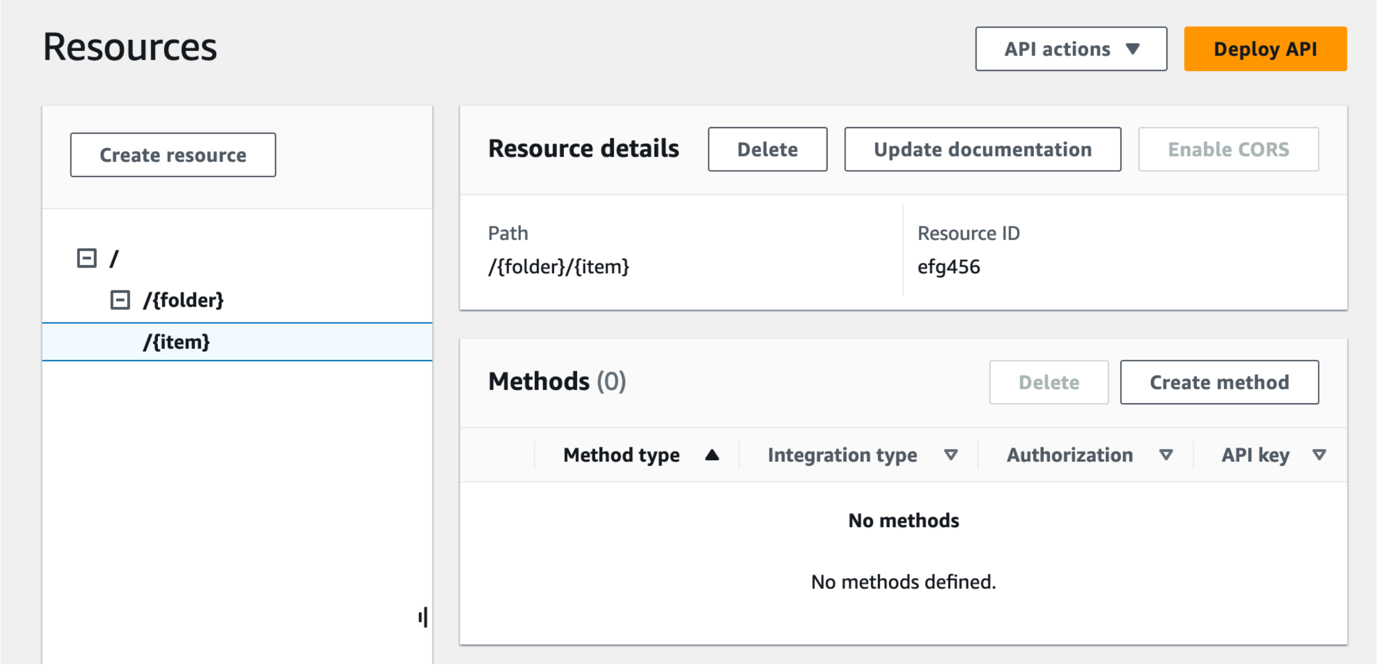
You use the API's root (/) resource as the container of an authenticated caller's Amazon S3 buckets. You also create a `Folder` and `Item` resources to represent a particular Amazon S3 bucket and a particular Amazon S3 object, respectively. The folder name and object key will be specified, in the form of path parameters as part of a request URL, by the caller.

> **Note**
>
> When accessing objects whose object key includes / or any other special character, the character needs to be URL encoded. For example, `test/test.txt` should be encoded to `test%2Ftest.txt`.

To create an API resource that exposes the Amazon S3 service features

1. In the same AWS Region you created your Amazon S3 bucket, create an API named **MyS3**. This API's root resource (**/**) represents the Amazon S3 service. In this step, you create two additional resources **/{folder}** and **/{item}**.
2. Select the API's root resource, and then choose **Create resource**.
3. Keep **Proxy resource** turned off.
4. For **Resource path**, select /.
5. For **Resource name**, enter **`{folder}`**.
6. Keep **CORS (Cross Origin Resource Sharing)** unchecked.
7. Choose **Create resource**.
8. Select the **/{folder}** resource, and then choose **Create resource**.
9. Use the previous steps to create a child resource of **/{folder}** named **`{item}`**.

   Your final API should look similar to the following:



Path
/{folder}/{item}

Resource ID
efg456

No methods

No methods defined.

Expose an API method to list the caller's Amazon S3 buckets

Getting the list of Amazon S3 buckets of the caller involves invoking the GET Service action on Amazon S3. On the API's root resource, (**/**), create the GET method. Configure the GET method to integrate with the Amazon S3, as follows.

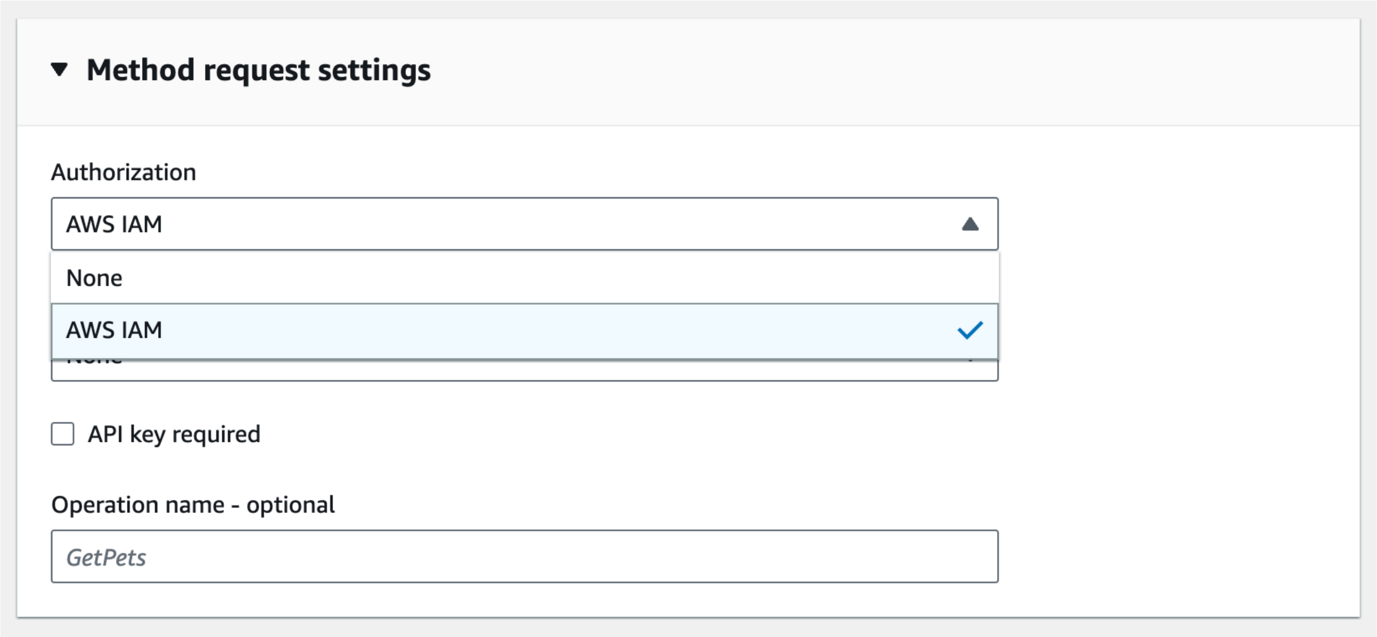
To create and initialize the API's GET / method

1. Select the **/** resource, and then choose **Create method**.
2. For method type, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Amazon S3 bucket.
5. For **AWS service**, select **Amazon Simple Storage Service**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **GET**.
8. For **Action type**, select **Use path override**.

   With path override, API Gateway forwards the client request to Amazon S3 as the corresponding Amazon S3 REST API path-style request, in which a Amazon S3 resource is expressed by the resource path of the `s3-host-name/bucket/key` pattern. API Gateway sets the `s3-host-name` and passes the client specified `bucket` and `key` from the client to Amazon S3.

9. For **Path override**, enter **/**.
10. For **Execution role**, enter the role ARN for **`APIGatewayS3ProxyPolicy`**.
11. Choose **Method request settings**.

    You use the method request settings to control who can call this method of your API.
12. For **Authorization**, from the dropdown menu, select `AWS_IAM`.



▼ **Method request settings**

Authorization

AWS IAM ▲

None

AWS IAM ✓

□ API key required

Operation name - optional

GetPets

13. Choose **Create method**.

This setup integrates the frontend `GET https://`*`your-api-host`*`/`*`stage`*`/` request with the backend `GET https://`*`your-s3-host`*`/`.

For your API to return successful responses and exceptions properly to the caller, you declare the 200, 400 and 500 responses in **Method response**. You use the default mapping for 200 responses so that backend responses of the status code not declared here will be returned to the caller as 200 ones.

To declare response types for the GET / method

1. On the **Method response** tab, under **Response 200**, choose **Edit**.
2. Choose **Add header** and do the following:

   a. For **Header name**, enter **`Content-Type`**.

   b. Choose **Add header**.

   Repeat these steps to create a **`Timestamp`** header and a **`Content-Length`** header.

3. Choose **Save**.
4. On the **Method response** tab, under **Method responses**, choose **Create response**.
5. For **HTTP status code**, enter **400**.

   You do not set any headers for this response.

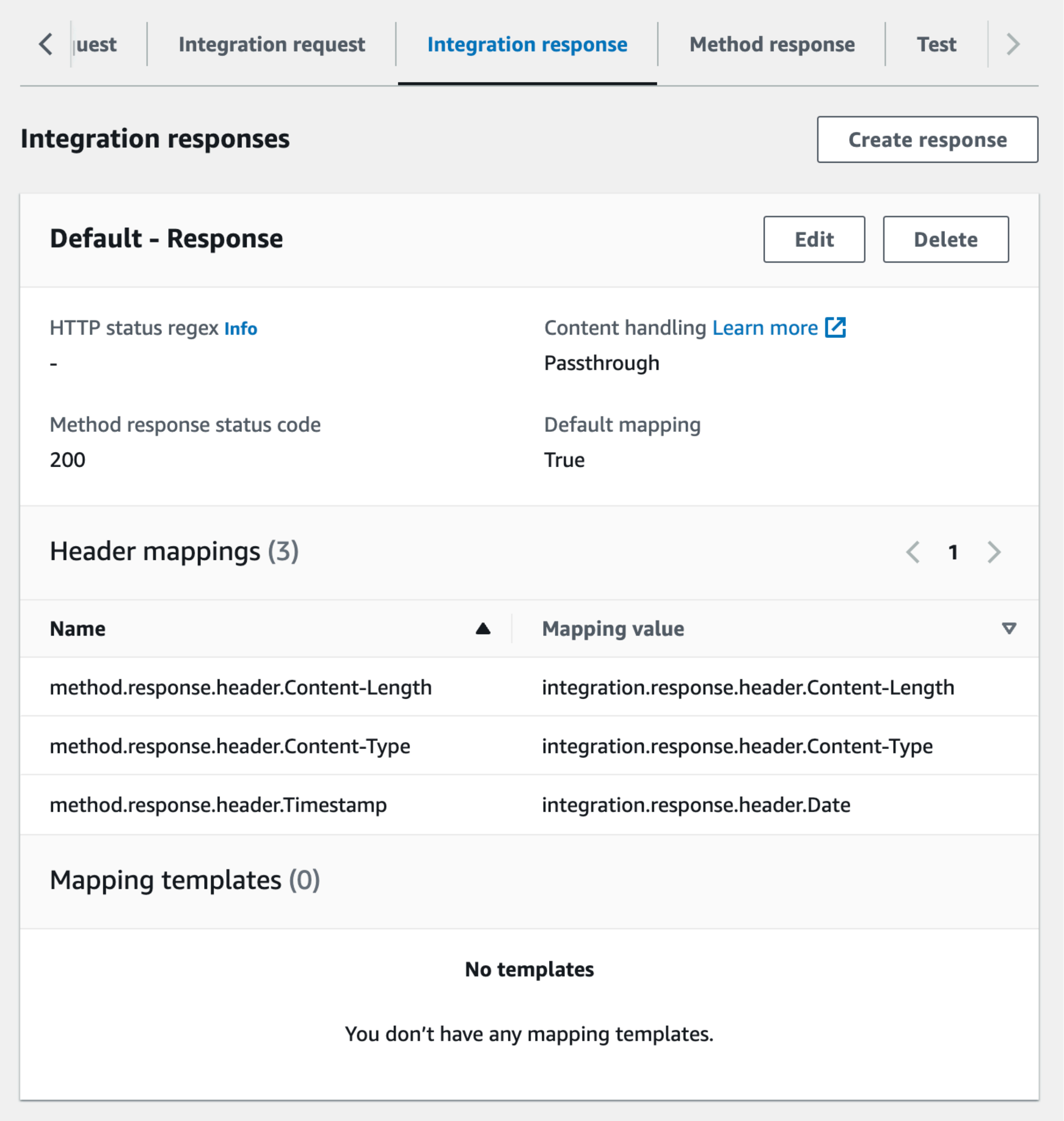
6. Choose **Save**.
7. Repeat the following steps to create the 500 response.

   You do not set any headers for this response.

Because the successful integration response from Amazon S3 returns the bucket list as an XML payload and the default method response from API Gateway returns a JSON payload, you must map the backend Content-Type header parameter value to the frontend counterpart. Otherwise, the client will receive `application/json` for the content type when the response body is actually an XML string. The following procedure shows how to set this up. In addition, you also want to display to the client other header parameters, such as Date and Content-Length.

To set up response header mappings for the GET / method

1. On the **Integration response** tab, under **Default - Response**, choose **Edit**.
2. For the **Content-Length** header, enter **`integration.response.header.Content-Length`** for the mapping value.
3. For the **Content-Type** header, enter **`integration.response.header.Content-Type`** for the mapping value.
4. For the **Timestamp** header, enter **`integration.response.header.Date`** for the mapping value.
5. Choose **Save**. The result should look similar to the following:



Integration responses

Default - Response

HTTP status regex Info
-

Content handling Learn more
Passthrough

Method response status code
200

Default mapping
True

Header mappings (3)

Name	Mapping value
method.response.header.Content-Length	integration.response.header.Content-Length
method.response.header.Content-Type	integration.response.header.Content-Type
method.response.header.Timestamp	integration.response.header.Date

Mapping templates (0)

No templates

You don’t have any mapping templates.

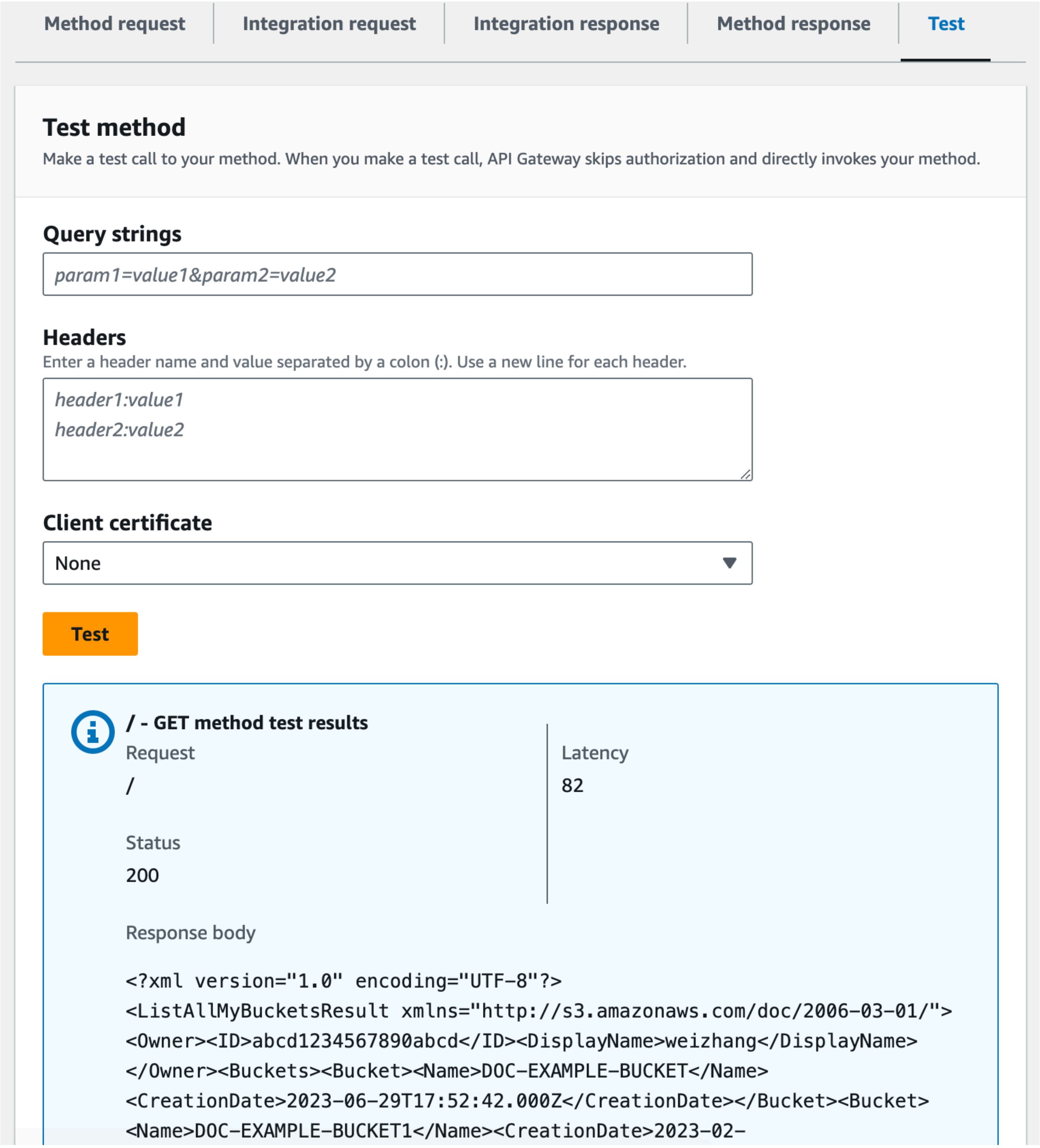
6. On the **Integration response** tab, under **Integration responses**, choose **Create response**.
7. For **HTTP status regex**, enter **`4\d{2}`**. This maps all 4xx HTTP response status codes to the method response.
8. For **Method response status code**, select **`400`**.
9. Choose **Create**.

10. Repeat the following steps to create an integration response for the 500 method response. For **HTTP status regex**, enter **`5\d{2}`**.

As a good practice, you can test the API you have configured so far.

To test the GET / method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. Choose **Test**. The result should look like the following image:



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/ - GET method test results

Request

/

Status

200

Latency

82

Response body

```
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
<Owner><ID>abcd1234567890abcd</ID><DisplayName>weizhang</DisplayName>
</Owner><Buckets><Bucket><Name>DOC-EXAMPLE-BUCKET</Name>
<CreationDate>2023-06-29T17:52:42.000Z</CreationDate></Bucket><Bucket>
<Name>DOC-EXAMPLE-BUCKET1</Name><CreationDate>2023-02-
```

Expose API methods to access an Amazon S3 bucket

To work with an Amazon S3 bucket, you expose the GET method on the /{folder} resource to list objects in a bucket. The instructions are similar to those described in Expose an API method to list the caller's Amazon S3 buckets. For more methods, you can import the sample API here, OpenAPI definitions of the sample API as an Amazon S3 proxy.

To expose the GET method on a folder resource

1. Select the **/{folder}** resource, and then choose **Create method**.
2. For method type, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Amazon S3 bucket.
5. For **AWS service**, select **Amazon Simple Storage Service**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **GET**.
8. For **Action type**, select **Use path override**.
9. For **Path override**, enter **`{bucket}`**.
10. For **Execution role**, enter the role ARN for **`APIGatewayS3ProxyPolicy`**.
11. Choose **Create method**.

You set the `{folder}` path parameter in the Amazon S3 endpoint URL. You need to map the `{folder}` path parameter of the method request to the `{bucket}` path parameter of the integration request.

To map `{folder}` to `{bucket}`

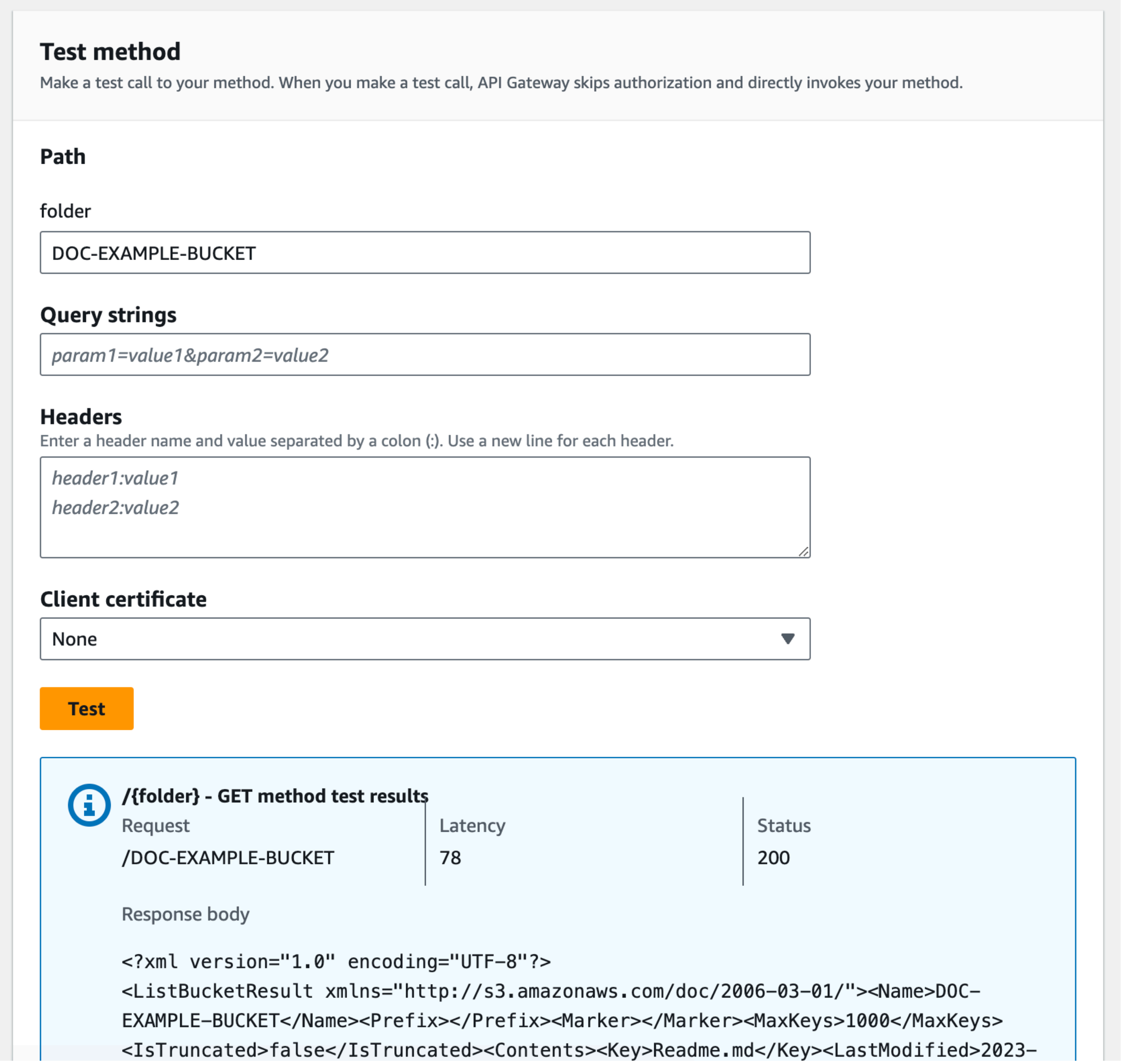
1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. Choose **URL path parameters**, and then choose **Add path parameter**.
3. For **Name**, enter **`bucket`**.
4. For **Mapped from**, enter **`method.request.path.folder`**.
5. Choose **Save**.

Now, you test your API.

To test the `/{folder}` `GET` method.

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. Under **Path**, for **folder**, enter the name of your bucket.
3. Choose **Test**.

   The test result will contain a list of object in your bucket.



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

folder: DOC-EXAMPLE-BUCKET

Query strings: *param1=value1&param2=value2*

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate: None

/{folder} - GET method test results

Request	Latency	Status
/DOC-EXAMPLE-BUCKET	78	200

Response body

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/"><Name>DOC-
EXAMPLE-BUCKET</Name><Prefix></Prefix><Marker></Marker><MaxKeys>1000</MaxKeys>
<IsTruncated>false</IsTruncated><Contents><Key>Readme.md</Key><LastModified>2023-
```

Expose API methods to access an Amazon S3 object in a bucket

Amazon S3 supports GET, DELETE, HEAD, OPTIONS, POST and PUT actions to access and manage objects in a given bucket. In this tutorial, you expose a GET method on the `{folder}/{item}` resource to get an image from a bucket. For more applications of the `{folder}/{item}` resource, see the sample API, [OpenAPI definitions of the sample API as an Amazon S3 proxy](#).

To expose the GET method on a item resource

1. Select the **/{item}** resource, and then choose **Create method**.
2. For method type, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Amazon S3 bucket.
5. For **AWS service**, select **Amazon Simple Storage Service**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, select **GET**.
8. For **Action type**, select **Use path override**.
9. For **Path override**, enter **{bucket}/{object}**.
10. For **Execution role**, enter the role ARN for **`APIGatewayS3ProxyPolicy`**.
11. Choose **Create method**.

You set the `{folder}` and `{item}` path parameters in the Amazon S3 endpoint URL. You need to map the path parameter of the method request to the path parameter of the integration request.

In this step, you do the following:

- Map the `{folder}` path parameter of the method request to the `{bucket}` path parameter of the integration request.
- Map the `{item}` path parameter of the method request to the `{object}` path parameter of the integration request.

To map `{folder}` to `{bucket}` and `{item}` to `{object}`

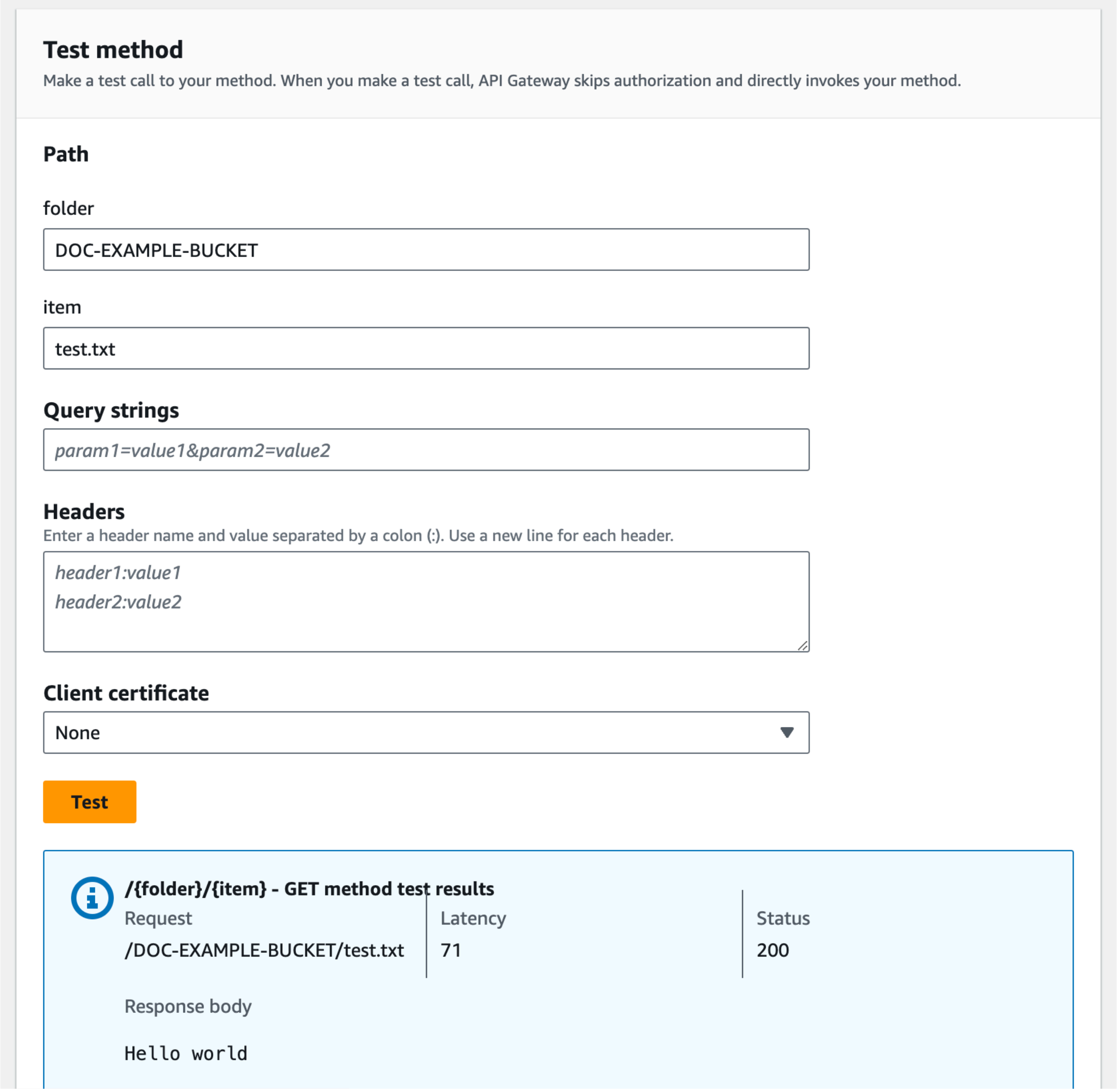
1. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
2. Choose **URL path parameters**.

3. Choose **Add path parameter**.
4. For **Name**, enter **`bucket`**.
5. For **Mapped from**, enter **`method.request.path.folder`**.
6. Choose **Add path parameter**.
7. For **Name**, enter **`object`**.
8. For **Mapped from**, enter **`method.request.path.item`**.
9. Choose **Save**.

To test the `/{folder}/{object} GET` method.

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. Under **Path**, for **folder**, enter the name of your bucket.
3. Under **Path**, for **item**, enter the name of an item.
4. Choose **Test**.

   The response body will contain the contents of the item.



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

folder

DOC-EXAMPLE-BUCKET

item

test.txt

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/{folder}/{item} - GET method test results

Request	Latency	Status
/DOC-EXAMPLE-BUCKET/test.txt	71	200

Response body

```
Hello world
```

The request correctly returns the plain text of ("Hello world") as the content of the specified file (test.txt) in the given Amazon S3 bucket (DOC-EXAMPLE-BUCKET).

To download or upload binary files, which in API Gateway is considered any thing other than utf-8 encoded JSON content, additional API settings are necessary. This is outlined as follows:

To download or upload binary files from S3

1. Register the media types of the affected file to the API's binaryMediaTypes. You can do this in the console:

    a. Choose **API settings** for the API.

    b. Under **Binary media types**, choose **Manage media types**.

    c. Choose **Add binary media type**, and then enter the required media type, for example, `image/png`.

    d. Choose **Save changes** to save the setting.

2. Add the `Content-Type` (for upload) and/or `Accept` (for download) header to the method request to require the client to specify the required binary media type and map them to the integration request.
3. Set **Content Handling** to `Passthrough` in the integration request (for upload) and in a integration response (for download). Make sure that no mapping template is defined for the affected content type. For more information, see Integration Passthrough Behaviors and Select VTL Mapping Templates.

The payload size limit is 10 MB. See API Gateway quotas for configuring and running a REST API.

Make sure that files on Amazon S3 have the correct content types added as the files' metadata. For streamable media content, `Content-Disposition:inline` may also need to be added to the metadata.

For more information about the binary support in API Gateway, see Content type conversions in API Gateway.

OpenAPI definitions of the sample API as an Amazon S3 proxy

The following OpenAPI definitions describes an API that works as an Amazon S3 proxy. This API contains more Amazon S3 operations than the API you created in the tutorial. The following methods are exposed in the OpenAPI definitions:

- Expose GET on the API's root resource to list all of the Amazon S3 buckets of a caller.
- Expose GET on a Folder resource to view a list of all of the objects in an Amazon S3 bucket.
- Expose PUT on a Folder resource to add a bucket to Amazon S3.
- Expose DELETE on a Folder resource to remove a bucket from Amazon S3.

- Expose GET on a Folder/Item resource to view or download an object from an Amazon S3 bucket.
- Expose PUT on a Folder/Item resource to upload an object to an Amazon S3 bucket.
- Expose HEAD on a Folder/Item resource to get object metadata in an Amazon S3 bucket.
- Expose DELETE on a Folder/Item resource to remove an object from an Amazon S3 bucket.

For instructions on how to import an API using the OpenAPI definition, see Configuring a REST API using OpenAPI.

For instructions on how to create a similar API, see Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway.

To learn how to invoke this API using Postman, which supports the AWS IAM authorization, see Call the API using a REST API client.

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-13T23:04:43Z",
    "title": "MyS3"
  },
  "host": "9gn28ca086.execute-api.{region}.amazonaws.com",
  "basePath": "/S3",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Timestamp": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length":
 "integration.response.header.Content-Length",
                "method.response.header.Timestamp":
 "integration.response.header.Date"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path//",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      }
    },
    "/{folder}": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Date": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Date": "integration.response.header.Date",
                "method.response.header.Content-Length":
 "integration.response.header.content-length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
```

```
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length":
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder",
            "integration.request.header.Content-Type":
 "method.request.header.Content-Type"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "PUT",
          "type": "aws"
        }
      },
      "delete": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Date": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
"integration.response.header.Content-Type",
                "method.response.header.Date": "integration.response.header.Date"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "DELETE",
          "type": "aws"
        }
      }
    },
    "/{folder}/{item}": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "content-type": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.content-type":
"integration.response.header.content-type",
                "method.response.header.Content-Type":
"integration.response.header.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "head": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
"integration.response.header.Content-Type",
                "method.response.header.Content-Length":
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "HEAD",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
          },
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length":
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder",
            "integration.request.header.Content-Type":
 "method.request.header.Content-Type"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "PUT",
          "type": "aws"
        }
      },
      "delete": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200"
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "DELETE",
          "type": "aws"
        }
      }
    }
  },
  "securityDefinitions": {
    "sigv4": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "awsSigv4"
    }
  },
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "MyS3",
    "version" : "2016-10-13T23:04:43Z"
  },
  "servers" : [ {
    "url" : "https://9gn28ca086.execute-api.{region}.amazonaws.com/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "S3"
      }
    }
  } ],
  "paths" : {
    "/{folder}" : {
      "get" : {
        "parameters" : [ {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Date" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "GET",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type",
                "method.response.header.Date" : "integration.response.header.Date",
                "method.response.header.Content-Length" :
 "integration.response.header.content-length"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "put" : {
        "parameters" : [ {
          "name" : "Content-Type",
          "in" : "header",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "PUT",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" : 
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length" : 
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.bucket" : "method.request.path.folder",
            "integration.request.header.Content-Type" : 
 "method.request.header.Content-Type"
```

```
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "delete" : {
        "parameters" : [ {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Date" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "DELETE",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type",
                "method.response.header.Date" : "integration.response.header.Date"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      }
    },
    "/{folder}/{item}" : {
      "get" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "content-type" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "GET",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.content-type" :
 "integration.response.header.content-type",
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "put" : {
        "parameters" : [ {
          "name" : "Content-Type",
          "in" : "header",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "PUT",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length" :
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder",
            "integration.request.header.Content-Type" :
 "method.request.header.Content-Type"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "delete" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "DELETE",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200"
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
```

```
      },
      "head" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "HEAD",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length" :
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      }
    },
    "/" : {
      "get" : {
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Timestamp" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "GET",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path//",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
                "method.response.header.Content-Length" :
 "integration.response.header.Content-Length",
                "method.response.header.Timestamp" :
 "integration.response.header.Date"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "Empty" : {
        "title" : "Empty Schema",
        "type" : "object"
      }
    }
  }
}
```

Call the API using a REST API client

To provide an end-to-end tutorial, we now show how to call the API using Postman, which supports the AWS IAM authorization.

To call our Amazon S3 proxy API using Postman

1. Deploy or redeploy the API. Make a note of the base URL of the API that is displayed next to **Invoke URL** at the top of the **Stage Editor**.
2. Launch Postman.
3. Choose **Authorization** and then choose `AWS Signature`. Type your IAM user's Access Key ID and Secret Access Key into the **AccessKey** and **SecretKey**input fields, respectively. Type the AWS region to which your API is deployed in the **AWS Region** text box. Type `execute-api` in the **Service Name** input field.

You can create a pair of the keys from the **Security Credentials** tab from your IAM user account in the IAM Management Console.

4. To add a bucket named `apig-demo-5` to your Amazon S3 account in the *{region}* region:

   > **Note**
   >
   > Be sure that the bucket name must be globally unique.

   a. Choose **PUT** from the drop-down method list and type the method URL (`https://`*api-id*`.execute-api.`*aws-region*`.amazonaws.com/`*stage*`/`*folder-name*

   b. Set the `Content-Type` header value as `application/xml`. You may need to delete any existing headers before setting the content type.

   c. Choose **Body** menu item and type the following XML fragment as the request body:

   ```
   <CreateBucketConfiguration>
     <LocationConstraint>{region}</LocationConstraint>
   </CreateBucketConfiguration>
   ```

   d. Choose **Send** to submit the request. If successful, you should receive a `200 OK` response with an empty payload.

5. To add a text file to a bucket, follow the instructions above. If you specify a bucket name of **`apig-demo-5`** for `{folder}` and a file name of **`Readme.txt`** for `{item}` in the URL and provide a text string of **`Hello, World!`** as the file contents (thereby making it the request payload), the request becomes

```
PUT /S3/apig-demo-5/Readme.txt HTTP/1.1
Host: 9gn28ca086.execute-api.{region}.amazonaws.com
Content-Type: application/xml
X-Amz-Date: 20161015T062647Z
Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/execute-
api/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=ccadb877bdb0d395ca38cc47e18a0d76bb5eaf17007d11e40bf6fb63d28c705b
Cache-Control: no-cache
Postman-Token: 6135d315-9cc4-8af8-1757-90871d00847e

Hello, World!
```

If everything goes well, you should receive a `200 OK` response with an empty payload.

6. To get the content of the `Readme.txt` file we just added to the `apig-demo-5` bucket, do a GET request like the following one:

   ```
   GET /S3/apig-demo-5/Readme.txt HTTP/1.1
   Host: 9gn28ca086.execute-api.{region}.amazonaws.com
   Content-Type: application/xml
   X-Amz-Date: 20161015T063759Z
   Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/
   execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
    Signature=ba09b72b585acf0e578e6ad02555c00e24b420b59025bc7bb8d3f7aed1471339
   Cache-Control: no-cache
   Postman-Token: d60fcb59-d335-52f7-0025-5bd96928098a
   ```

   If successful, you should receive a `200 OK` response with the `Hello, World!` text string as the payload.

7. To list items in the `apig-demo-5` bucket, submit the following request:

   ```
   GET /S3/apig-demo-5 HTTP/1.1
   Host: 9gn28ca086.execute-api.{region}.amazonaws.com
   Content-Type: application/xml
   X-Amz-Date: 20161015T064324Z
   Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/
   execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
    Signature=4ac9bd4574a14e01568134fd16814534d9951649d3a22b3b0db9f1f5cd4dd0ac
   Cache-Control: no-cache
   Postman-Token: 9c43020a-966f-61e1-81af-4c49ad8d1392
   ```

   If successful, you should receive a `200 OK` response with an XML payload showing a single item in the specified bucket, unless you added more files to the bucket before submitting this request.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
       <Name>apig-demo-5</Name>
       <Prefix></Prefix>
       <Marker></Marker>
       <MaxKeys>1000</MaxKeys>
       <IsTruncated>false</IsTruncated>
   ```

```
    <Contents>
        <Key>Readme.txt</Key>
        <LastModified>2016-10-15T06:26:48.000Z</LastModified>
        <ETag>"65a8e27d8879283831b664bd8b7f0ad4"</ETag>
        <Size>13</Size>
        <Owner>
            <ID>06e4b09e9d...603addd12ee</ID>
            <DisplayName>user-name</DisplayName>
        </Owner>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

Note

To upload or download an image, you need to set content handling to CONVERT_TO_BINARY.

Tutorial: Create a REST API as an Amazon Kinesis proxy in API Gateway

This page describes how to create and configure a REST API with an integration of the AWS type to access Kinesis.

Note

To integrate your API Gateway API with Kinesis, you must choose a region where both the API Gateway and Kinesis services are available. For region availability, see Service Endpoints and Quotas.

For the purpose of illustration, we create an example API to enable a client to do the following:

1. List the user's available streams in Kinesis
2. Create, describe, or delete a specified stream
3. Read data records from or write data records into the specified stream

To accomplish the preceding tasks, the API exposes methods on various resources to invoke the following, respectively:

1. The `ListStreams` action in Kinesis
2. The `CreateStream`, `DescribeStream`, or `DeleteStream` action
3. The `GetRecords` or `PutRecords` (including `PutRecord`) action in Kinesis

Specifically, we build the API as follows:

- Expose an HTTP GET method on the API's `/streams` resource and integrate the method with the ListStreams action in Kinesis to list the streams in the caller's account.
- Expose an HTTP POST method on the API's `/streams/{stream-name}` resource and integrate the method with the CreateStream action in Kinesis to create a named stream in the caller's account.
- Expose an HTTP GET method on the API's `/streams/{stream-name}` resource and integrate the method with the DescribeStream action in Kinesis to describe a named stream in the caller's account.
- Expose an HTTP DELETE method on the API's `/streams/{stream-name}` resource and integrate the method with the DeleteStream action in Kinesis to delete a stream in the caller's account.
- Expose an HTTP PUT method on the API's `/streams/{stream-name}/record` resource and integrate the method with the PutRecord action in Kinesis. This enables the client to add a single data record to the named stream.
- Expose an HTTP PUT method on the API's `/streams/{stream-name}/records` resource and integrate the method with the PutRecords action in Kinesis. This enables the client to add a list of data records to the named stream.
- Expose an HTTP GET method on the API's `/streams/{stream-name}/records` resource and integrate the method with the GetRecords action in Kinesis. This enables the client to list data records in the named stream, with a specified shard iterator. A shard iterator specifies the shard position from which to start reading data records sequentially.
- Expose an HTTP GET method on the API's `/streams/{stream-name}/sharditerator` resource and integrate the method with the GetShardIterator action in Kinesis. This helper method must be supplied to the `ListStreams` action in Kinesis.

You can apply the instructions presented here to other Kinesis actions. For the complete list of the Kinesis actions, see Amazon Kinesis API Reference.

Instead of using the API Gateway console to create the sample API, you can import the sample API into API Gateway using the API Gateway Import API. For information on how to use the Import API, see Configuring a REST API using OpenAPI.

Create an IAM role and policy for the API to access Kinesis

To allow the API to invoke Kinesis actions, you must have the appropriate IAM policies attached to an IAM role.

To create the AWS service proxy execution role

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Roles**.
3. Choose **Create role**.
4. Choose **AWS service** under **Select type of trusted entity**, and then select **API Gateway** and select **Allows API Gateway to push logs to CloudWatch Logs**.
5. Choose **Next**, and then choose **Next**.
6. For **Role name**, enter **`APIGatewayKinesisProxyPolicy`**, and then choose **Create role**.
7. In the **Roles** list, choose the role you just created. You may need to scroll or use the search bar to find the role.
8. For the selected role, select the **Add permissions** tab.
9. Choose **Attach policies** from the dropdown list.
10. In the search bar, enter **`AmazonKinesisFullAccess`** and choose **Add permissions**.

    Note

    This tutorial uses a managed policy for simplicity. As a best practice, you should create your own IAM policy to grant the minimum permissions required.

11. Note the newly created **Role ARN**, you will use it later.

Start to create an API as a Kinesis proxy

Use the following steps to create the API in the API Gateway console.

To create an API as an AWS service proxy for Kinesis

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

   If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.
3. Choose **New API**.
4. In **API name**, enter **`KinesisProxy`**. Keep the default values for all other fields.
5. (Optional) For **Description**, enter a description.
6. Choose **Create API**.

After the API is created, the API Gateway console displays the **Resources** page, which contains only the API's root (/) resource.

List streams in Kinesis

Kinesis supports the `ListStreams` action with the following REST API call:

```
POST /?Action=ListStreams HTTP/1.1
Host: kinesis.<region>.<domain>
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.1
Authorization: <AuthParams>
X-Amz-Date: <Date>

{
   ...
}
```

In the above REST API request, the action is specified in the `Action` query parameter. Alternatively, you can specify the action in a `X-Amz-Target` header, instead:

```
POST / HTTP/1.1
Host: kinesis.<region>.<domain>
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.1
Authorization: <AuthParams>
X-Amz-Date: <Date>
X-Amz-Target: Kinesis_20131202.ListStreams
{
   ...
}
```

In this tutorial, we use the query parameter to specify action.

To expose a Kinesis action in the API, add a `/streams` resource to the API's root. Then set a GET method on the resource and integrate the method with the `ListStreams` action of Kinesis.

The following procedure describes how to list Kinesis streams by using the API Gateway console.

To list Kinesis streams by using the API Gateway console

1. Select the / resource, and then choose **Create resource**.
2. For **Resource name**, enter **`streams`**.
3. Keep **CORS (Cross Origin Resource Sharing)** turned off.
4. Choose **Create resource**.
5. Choose the `/streams` resource, and then choose **Create method**, and then do the following:

   a. For **Method type**, select **GET**.

   > **Note**
   >
   > The HTTP verb for a method invoked by a client may differ from the HTTP verb for an integration required by the backend. We select GET here, because listing streams is intuitively a READ operation.

   b. For **Integration type**, select **AWS service**.

   c. For **AWS Region**, select the AWS Region where you created your Kinesis stream.

   d. For **AWS service**, select **Kinesis**.

   e. Keep **AWS subdomain** blank.

f. For **HTTP method**, choose **POST**.

> **Note**
>
> We chose POST here because Kinesis requires that the `ListStreams` action be invoked with it.

g. For **Action type**, choose **Use action name**.

h. For **Action name**, enter **`ListStreams`**.

i. For **Execution role**, enter the ARN for your execution role.

j. Keep the default of **Passthrough** for **Content Handling**.

k. Choose **Create method**.

6. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
7. For **Request body passthrough**, select **When there are no templates defined (recommended)**.
8. Choose **URL request headers parameters**, and then do the following:

   a. Choose **Add request headers parameter**.

   b. For **Name**, enter **`Content-Type`**.

   c. For **Mapped from**, enter **`'application/x-amz-json-1.1'`**.

   We use a request parameter mapping to set the `Content-Type` header to the static value of `'application/x-amz-json-1.1'` to inform Kinesis that the input is of a specific version of JSON.
9. Choose **Mapping templates**, and then choose **Add mapping template**, and do the following:

   a. For **Content-Type**, enter **`application/json`**.

   b. For **Template body**, enter **`{}`**.

   c. Choose **Save**.

   The ListStreams request takes a payload of the following JSON format:

```
{
    "ExclusiveStartStreamName": "string",
    "Limit": number
}
```

However, the properties are optional. To use the default values, we opted for an empty JSON payload here.

10. Test the GET method on the **/streams** resource to invoke the `ListStreams` action in Kinesis:

    Choose the **Test** tab. You might need to choose the right arrow button to show the tab.

    Choose **Test** to test your method.

    If you already created two streams named "myStream" and "yourStream" in Kinesis, the successful test returns a 200 OK response containing the following payload:

    ```
    {
         "HasMoreStreams": false,
         "StreamNames": [
             "myStream",
             "yourStream"
         ]
    }
    ```

Create, describe, and delete a stream in Kinesis

Creating, describing, and deleting a stream in Kinesis involves making the following Kinesis REST API requests, respectively:

```
POST /?Action=CreateStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardCount": number,
    "StreamName": "string"
}
```

```
POST /?Action=DescribeStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "StreamName": "string"
}
```

```
POST /?Action=DeleteStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "StreamName":"string"
}
```

We can build the API to accept the required input as a JSON payload of the method request and pass the payload through to the integration request. However, to provide more examples of data mapping between method and integration requests, and method and integration responses, we create our API somewhat differently.

We expose the `GET`, `POST`, and `Delete` HTTP methods on a to-be-named `Stream` resource. We use the `{stream-name}` path variable as the placeholder of the stream resource and integrate these API methods with the Kinesis' `DescribeStream`, `CreateStream`, and `DeleteStream` actions, respectively. We require that the client pass other input data as headers, query parameters, or the payload of a method request. We provide mapping templates to transform the data to the required integration request payload.

To create the {stream-name} resource

1. Choose the **/streams** resource, and then choose **Create resource**.

2. Keep **Proxy resource** turned off.
3. For **Resource path**, select `/streams`.
4. For **Resource name**, enter **`{stream-name}`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.

To configure and test the GET method on a stream resource

1. Choose the **/{stream-name}** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`DescribeStream`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **Create method**.
13. In the **Integration request** section, add the following **URL request headers parameters**:

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    The task follows the same procedure to set up the request parameter mapping for the `GET /streams` method.
14. Add the following body mapping template to map data from the `GET /streams/{stream-name}` method request to the `POST /?Action=DescribeStream` integration request:

    ```
    {
        "StreamName": "$input.params('stream-name')"
    }
    ```

This mapping template generates the required integration request payload for the `DescribeStream` action of Kinesis from the method request's `stream-name` path parameter value.

15. To test the `GET /stream/{stream-name}` method to invoke the `DescribeStream` action in Kinesis, choose the **Test** tab.

16. For **Path**, under **stream-name**, enter the name of an existing Kinesis stream.

17. Choose **Test**. If the test is successful, a 200 OK response is returned with a payload similar to the following:

```
{
  "StreamDescription": {
    "HasMoreShards": false,
    "RetentionPeriodHours": 24,
    "Shards": [
      {
        "HashKeyRange": {
          "EndingHashKey": "68056473384187692692674921486353642290",
          "StartingHashKey": "0"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
 "49559266461454070523309915164834022007924120923395850242"
        },
        "ShardId": "shardId-000000000000"
      },
      ...
      {
        "HashKeyRange": {
          "EndingHashKey": "340282366920938463463374607431768211455",
          "StartingHashKey": "272225893536750770770699685945414569164"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
 "49559266461543273504104037657400164881014714369419771970"
        },
        "ShardId": "shardId-000000000004"
      }
    ],
    "StreamARN": "arn:aws:kinesis:us-east-1:12345678901:stream/myStream",
    "StreamName": "myStream",
    "StreamStatus": "ACTIVE"
  }
}
```

After you deploy the API, you can make a REST request against this API method:

```
GET https://your-api-id.execute-api.region.amazonaws.com/stage/streams/myStream
 HTTP/1.1
Host: your-api-id.execute-api.region.amazonaws.com
Content-Type: application/json
Authorization: ...
X-Amz-Date: 20160323T194451Z
```

To configure and test the POST method on a stream resource

1. Choose the **/{stream-name}** resource, and then choose **Create method**.
2. For **Method type**, select **POST**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`CreateStream`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **Create method**.
13. In the **Integration request** section, add the following **URL request headers parameters**:

```
Content-Type: 'x-amz-json-1.1'
```

The task follows the same procedure to set up the request parameter mapping for the `GET /streams` method.

14. Add the following body mapping template to map data from the `POST /streams/{stream-name}` method request to the `POST /?Action=CreateStream` integration request:

    ```
    {
        "ShardCount": #if($input.path('$.ShardCount') == '') 5 #else
     $input.path('$.ShardCount') #end,
        "StreamName": "$input.params('stream-name')"
    }
    ```

    In the preceding mapping template, we set `ShardCount` to a fixed value of 5 if the client does not specify a value in the method request payload.

15. To test the `POST /stream/{stream-name}` method to invoke the `CreateStream` action in Kinesis, choose the **Test** tab.
16. For **Path**, under **stream-name**, enter the name of a new Kinesis stream.
17. Choose **Test**. If the test is successful, a 200 OK response is returned with no data.

    After you deploy the API, you can also make a REST API request against the POST method on a Stream resource to invoke the `CreateStream` action in Kinesis:

    ```
    POST https://your-api-id.execute-api.region.amazonaws.com/stage/streams/yourStream
     HTTP/1.1
    Host: your-api-id.execute-api.region.amazonaws.com
    Content-Type: application/json
    Authorization: ...
    X-Amz-Date: 20160323T194451Z

    {
        "ShardCount": 5
    }
    ```

Configure and test the DELETE method on a stream resource

1. Choose the **/{stream-name}** resource, and then choose **Create method**.

2. For **Method type**, select **DELETE**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **DeleteStream**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **Create method**.
13. In the **Integration request** section, add the following **URL request headers parameters**:

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    The task follows the same procedure to set up the request parameter mapping for the `GET /streams` method.

14. Add the following body mapping template to map data from the `DELETE /streams/{stream-name}` method request to the corresponding integration request of `POST /?Action=DeleteStream`:

    ```
    {
        "StreamName": "$input.params('stream-name')"
    }
    ```

    This mapping template generates the required input for the `DELETE /streams/{stream-name}` action from the client-supplied URL path name of `stream-name`.

15. To test the `DELETE /stream/{stream-name}` method to invoke the `DeleteStream` action in Kinesis, choose the **Test** tab.
16. For **Path**, under **stream-name**, enter the name of an existing Kinesis stream.
17. Choose **Test**. If the test is successful, a 200 OK response is returned with no data.

    After you deploy the API, you can also make the following REST API request against the DELETE method on the Stream resource to call the `DeleteStream` action in Kinesis:

```
DELETE https://your-api-id.execute-api.region.amazonaws.com/stage/
streams/yourStream HTTP/1.1
Host: your-api-id.execute-api.region.amazonaws.com
Content-Type: application/json
Authorization: ...
X-Amz-Date: 20160323T194451Z

{}
```

Get records from and add records to a stream in Kinesis

After you create a stream in Kinesis, you can add data records to the stream and read the data from the stream. Adding data records involves calling the PutRecords or PutRecord action in Kinesis. The former adds multiple records whereas the latter adds a single record to the stream.

```
POST /?Action=PutRecords HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "Records": [
        {
            "Data": blob,
            "ExplicitHashKey": "string",
            "PartitionKey": "string"
        }
    ],
    "StreamName": "string"
}
```

or

```
POST /?Action=PutRecord HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "Data": blob,
    "ExplicitHashKey": "string",
    "PartitionKey": "string",
    "SequenceNumberForOrdering": "string",
    "StreamName": "string"
}
```

Here, `StreamName` identifies the target stream to add records. `StreamName`, `Data`, and `PartitionKey` are required input data. In our example, we use the default values for all of the optional input data and will not explicitly specify values for them in the input to the method request.

Reading data in Kinesis amounts to calling the GetRecords action:

```
POST /?Action=GetRecords HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardIterator": "string",
    "Limit": number
}
```

Here, the source stream from which we are getting records is specified in the required `ShardIterator` value, as is shown in the following Kinesis action to obtain a shard iterator:

```

POST /?Action=GetShardIterator HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
```

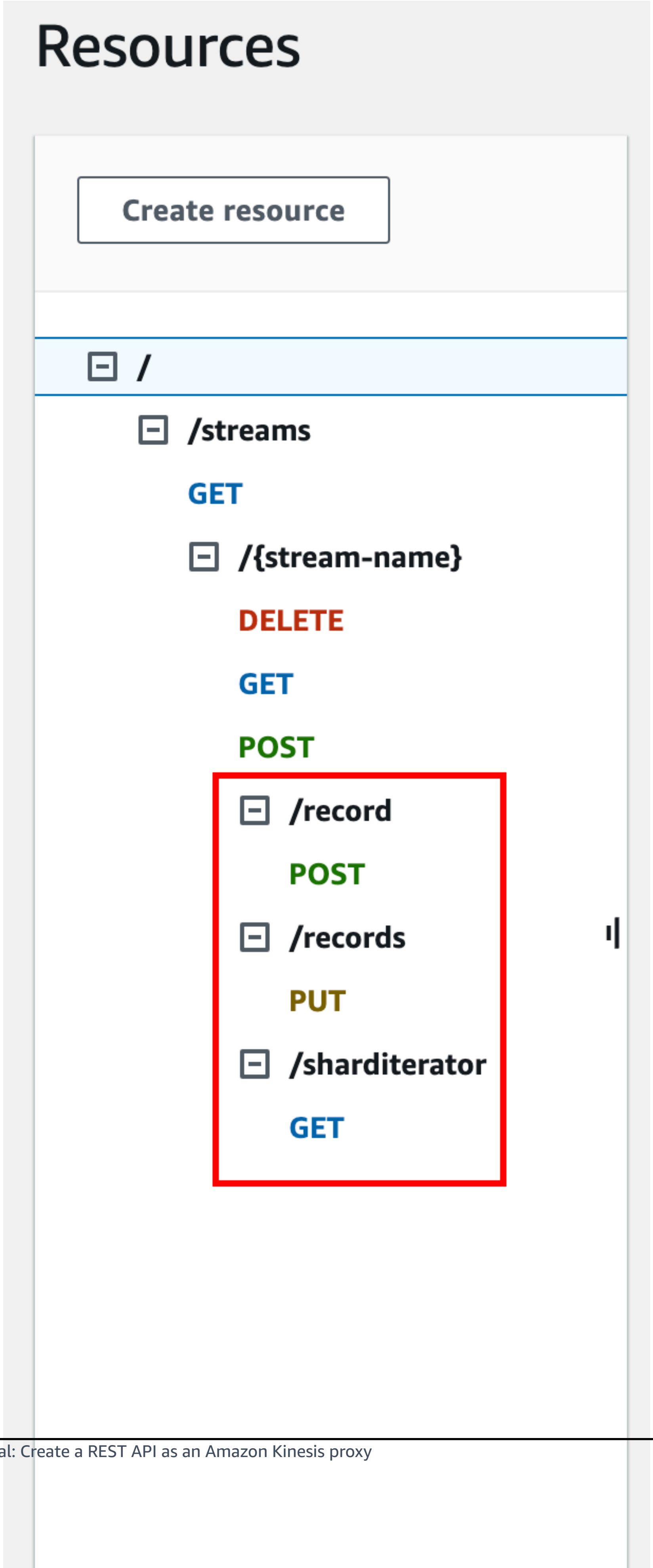
```
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardId": "string",
    "ShardIteratorType": "string",
    "StartingSequenceNumber": "string",
    "StreamName": "string"
}
```

For the `GetRecords` and `PutRecords` actions, we expose the GET and PUT methods, respectively, on a `/records` resource that is appended to a named stream resource (`/{stream-name}`). Similarly, we expose the `PutRecord` action as a PUT method on a `/record` resource.

Because the `GetRecords` action takes as input a `ShardIterator` value, which is obtained by calling the `GetShardIterator` helper action, we expose a GET helper method on a `ShardIterator` resource (`/sharditerator`).

To create the /record, /records, and /sharditerator resources

1. Choose the **/{stream-name}** resource, and then choose **Create resource**.
2. Keep **Proxy resource** turned off.
3. For **Resource path**, select `/{stream-name}`.
4. For **Resource name**, enter **`record`**.
5. Keep **CORS (Cross Origin Resource Sharing)** turned off.
6. Choose **Create resource**.
7. Repeat the previous steps to create a **/records** and a **/sharditerator** resource. The final API should look like the following:



Resources
Create resource
/
/streams
GET
/{stream-name}
DELETE
GET
POST
/record
POST
/records
PUT
/sharditerator
GET

The following four procedures describe how to set up each of the methods, how to map data from the method requests to the integration requests, and how to test the methods.

To set up and test the PUT `/streams/{stream-name}/record` method to invoke `PutRecord` in Kinesis:

1. Choose the **/record**, and then choose **Create method**.
2. For **Method type**, select **PUT**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`PutRecord`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **Create method**.
13. In the **Integration request** section, add the following **URL request headers parameters**:

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    The task follows the same procedure to set up the request parameter mapping for the `GET /streams` method.

14. Add the following body mapping template to map data from the `PUT /streams/{stream-name}/record` method request to the corresponding integration request of `POST /?Action=PutRecord`:

    ```
    {
        "StreamName": "$input.params('stream-name')",
        "Data": "$util.base64Encode($input.json('$.Data'))",
        "PartitionKey": "$input.path('$.PartitionKey')"
    }
    ```

    This mapping template assumes that the method request payload is of the following format:

```
{
   "Data": "some data",
   "PartitionKey": "some key"
}
```

This data can be modeled by the following JSON schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PutRecord proxy single-record payload",
  "type": "object",
  "properties": {
      "Data": { "type": "string" },
      "PartitionKey": { "type": "string" }
  }
}
```

You can create a model to include this schema and use the model to facilitate generating the mapping template. However, you can generate a mapping template without using any model.

15. To test the PUT `/streams/{stream-name}/record` method, set the `stream-name` path variable to the name of an existing stream, supply a payload of the required format, and then submit the method request. The successful result is a `200 OK` response with a payload of the following format:

```

{
  "SequenceNumber": "49559409944537880850133345460169886593573102115167928386",
  "ShardId": "shardId-000000000004"
}

```

To set up and test the PUT `/streams/{stream-name}/records` method to invoke `PutRecords` in Kinesis

1. Choose the **/records** resource, and then choose **Create method**.
2. For **Method type**, select **PUT**.
3. For **Integration type**, select **AWS service**.

4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`PutRecords`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **Create method**.
13. In the **Integration request** section, add the following **URL request headers parameters**:

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    The task follows the same procedure to set up the request parameter mapping for the `GET /streams` method.

14. Add the following mapping template to map data from the `PUT /streams/{stream-name}/records` method request to the corresponding integration request of `POST /?Action=PutRecords` :

    ```
    {
        "StreamName": "$input.params('stream-name')",
        "Records": [
           #foreach($elem in $input.path('$.records'))
              {
                "Data": "$util.base64Encode($elem.data)",
                "PartitionKey": "$elem.partition-key"
              }#if($foreach.hasNext),#end
            #end
        ]
    }
    ```

    This mapping template assumes that the method request payload can be modelled by the following JSON schema:

    ```
    {
      "$schema": "http://json-schema.org/draft-04/schema#",
      "title": "PutRecords proxy payload data",
    ```

```
  "type": "object",
  "properties": {
    "records": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "data": { "type": "string" },
          "partition-key": { "type": "string" }
        }
      }
    }
  }
}
```

You can create a model to include this schema and use the model to facilitate generating the mapping template. However, you can generate a mapping template without using any model.

In this tutorial, we used two slightly different payload formats to illustrate that an API developer can choose to expose the backend data format to the client or hide it from the client. One format is for the PUT `/streams/{stream-name}/records` method (above). Another format is used for the PUT `/streams/{stream-name}/record` method (in the previous procedure). In production environment, you should keep both formats consistent.

15. To test the PUT `/streams/{stream-name}/records` method, set the `stream-name` path variable to an existing stream, supply the following payload, and submit the method request.

```
{
    "records": [
        {
            "data": "some data",
            "partition-key": "some key"
        },
        {
            "data": "some other data",
            "partition-key": "some key"
        }
    ]
}
```

The successful result is a 200 OK response with a payload similar to the following output:

```
{
  "FailedRecordCount": 0,
  "Records": [
    {
      "SequenceNumber": "49559409944537880850133345460167468741933742152373764162",
      "ShardId": "shardId-000000000004"
    },
    {
      "SequenceNumber": "49559409944537880850133345460168677667753356781548470338",
      "ShardId": "shardId-000000000004"
    }
  ]
}
```

To set up and test the `GET /streams/{stream-name}/sharditerator` method invoke `GetShardIterator` in Kinesis

The `GET /streams/{stream-name}/sharditerator` method is a helper method to acquire a required shard iterator before calling the `GET /streams/{stream-name}/records` method.

1. Choose the **/sharditerator** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **AWS service**.
4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`GetShardIterator`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **URL query string parameters**.

The `GetShardIterator` action requires an input of a ShardId value. To pass a client-supplied `ShardId` value, we add a `shard-id` query parameter to the method request, as shown in the following step.

13. Choose **Add query string**.
14. For **Name**, enter **`shard-id`**.
15. Keep **Required** and **Caching** turned off.
16. Choose **Create method**.
17. In the **Integration request** section, add the following mapping template to generate the required input (`ShardId` and `StreamName`) to the `GetShardIterator` action from the `shard-id` and `stream-name` parameters of the method request. In addition, the mapping template also sets `ShardIteratorType` to `TRIM_HORIZON` as a default.

```
{
    "ShardId": "$input.params('shard-id')",
    "ShardIteratorType": "TRIM_HORIZON",
    "StreamName": "$input.params('stream-name')"
}
```

18. Using the **Test** option in the API Gateway console, enter an existing stream name as the `stream-name` **Path** variable value, set the `shard-id` **Query string** to an existing `ShardId` value (e.g., `shard-000000000004`), and choose **Test**.

    The successful response payload is similar to the following output:

```
{
  "ShardIterator": "AAAAAAAAAAFYVN3VlFy..."
}
```

    Make note of the `ShardIterator` value. You need it to get records from a stream.

To configure and test the `GET /streams/{stream-name}/records` method to invoke the `GetRecords` action in Kinesis

1. Choose the **/records** resource, and then choose **Create method**.
2. For **Method type**, select **GET**.
3. For **Integration type**, select **AWS service**.

4. For **AWS Region**, select the AWS Region where you created your Kinesis stream.
5. For **AWS service**, select **Kinesis**.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST**.
8. For **Action type**, choose **Use action name**.
9. For **Action name**, enter **`GetRecords`**.
10. For **Execution role**, enter the ARN for your execution role.
11. Keep the default of **Passthrough** for **Content Handling**.
12. Choose **HTTP request headers**.

    The `GetRecords` action requires an input of a `ShardIterator` value. To pass a client-supplied `ShardIterator` value, we add a `Shard-Iterator` header parameter to the method request.
13. Choose **Add header**.
14. For **Name**, enter **`Shard-Iterator`**.
15. Keep **Required** and **Caching** turned off.
16. Choose **Create method**.
17. In the **Integration request** section, add the following body mapping template to map the `Shard-Iterator` header parameter value to the `ShardIterator` property value of the JSON payload for the `GetRecords` action in Kinesis.

    ```
    {
        "ShardIterator": "$input.params('Shard-Iterator')"
    }
    ```
18. Using the **Test** option in the API Gateway console, enter an existing stream name as the `stream-name` **Path** variable value, set the `Shard-Iterator` **Header** to the `ShardIterator` value obtained from the test run of the `GET /streams/{stream-name}/sharditerator` method (above), and choose **Test**.

    The successful response payload is similar to the following output:

    ```
    {
      "MillisBehindLatest": 0,
      "NextShardIterator": "AAAAAAAAAF...",
      "Records": [ ... ]
    ```

```
}
```

OpenAPI definitions of a sample API as a Kinesis proxy

Following are OpenAPI definitions for the sample API as a Kinesis proxy used in this tutorial.

OpenAPI 3.0

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "KinesisProxy",
    "version": "2016-03-31T18:25:32Z"
  },
  "paths": {
    "/streams/{stream-name}/sharditerator": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "shard-id",
            "in": "query",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetShardIterator",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardId\": \"$input.params('shard-
id')\",\n    \"ShardIteratorType\": \"TRIM_HORIZON\",\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/records": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "Shard-Iterator",
            "in": "header",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardIterator\": \"$input.params('Shard-
Iterator')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "put": {
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/PutRecordsMethodRequestPayload"
              }
            },
            "application/x-amz-json-1.1": {
              "schema": {
                "$ref": "#/components/schemas/PutRecordsMethodRequestPayload"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Records\": [\n          {\n            \"Data\":
 \"$util.base64Encode($elem.data)\",\n            \"PartitionKey\":
 \"$elem.partition-key\"\n          }#if($foreach.hasNext),#end\n    ]\n}",
            "application/x-amz-json-1.1": "{\n  \"StreamName\":
 \"$input.params('stream-name')\",\n  \"records\" : [\n    {\n        \"Data
\" : \"$elem.data\",\n        \"PartitionKey\" : \"$elem.partition-key\"\n
  }#if($foreach.hasNext),#end\n  ]\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "post": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/CreateStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardCount\": 5,\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "delete": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Content-Type": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
```

```
           }
         },
         "400": {
           "description": "400 response",
           "headers": {
             "Content-Type": {
               "schema": {
                 "type": "string"
               }
             }
           },
           "content": {}
         },
         "500": {
           "description": "500 response",
           "headers": {
             "Content-Type": {
               "schema": {
                 "type": "string"
               }
             }
           },
           "content": {}
         }
       },
       "x-amazon-apigateway-integration": {
         "type": "aws",
         "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DeleteStream",
         "responses": {
           "4\\d{2}": {
             "statusCode": "400",
             "responseParameters": {
               "method.response.header.Content-Type":
"integration.response.header.Content-Type"
             }
           },
           "default": {
             "statusCode": "200",
             "responseParameters": {
               "method.response.header.Content-Type":
"integration.response.header.Content-Type"
             }
           },
           "5\\d{2}": {
             "statusCode": "500",
             "responseParameters": {
               "method.response.header.Content-Type":
"integration.response.header.Content-Type"
             }
           }
         },
         "requestParameters": {
           "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
         },
         "requestTemplates": {
           "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
         },
         "passthroughBehavior": "when_no_match",
         "httpMethod": "POST"
       }
     }
   },
   "/streams/{stream-name}/record": {
     "put": {
       "parameters": [
         {
           "name": "stream-name",
           "in": "path",
           "required": true,
           "schema": {
             "type": "string"
           }
         }
       ],
       "responses": {
         "200": {
           "description": "200 response",
           "content": {
             "application/json": {
               "schema": {
                 "$ref": "#/components/schemas/Empty"
               }
             }
           }
         }
       },
       "x-amazon-apigateway-integration": {
         "type": "aws",
         "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecord",
         "responses": {
           "default": {
             "statusCode": "200"
           }
         },
         "requestParameters": {
           "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
         },
         "requestTemplates": {
           "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Data\": \"$util.base64Encode($input.json('$.Data'))\",\n
\"PartitionKey\": \"$input.path('$.PartitionKey')\"\n}"
         },
         "passthroughBehavior": "when_no_match",
         "httpMethod": "POST"
       }
     }
   },
   "/streams": {
     "get": {
       "responses": {
         "200": {
           "description": "200 response",
           "content": {
             "application/json": {
               "schema": {
                 "$ref": "#/components/schemas/Empty"
               }
             }
           }
         }
       },
       "x-amazon-apigateway-integration": {
         "type": "aws",
         "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
         "responses": {
           "default": {
             "statusCode": "200"
           }
         },
         "requestParameters": {
           "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
         },
         "requestTemplates": {
           "application/json": "{\n}"
         },
         "passthroughBehavior": "when_no_match",
         "httpMethod": "POST"
       }
     }
   }
 },
 "components": {
   "schemas": {
     "Empty": {
       "type": "object"
     },
     "PutRecordsMethodRequestPayload": {
       "type": "object",
       "properties": {
         "records": {
           "type": "array",
           "items": {
             "type": "object",
             "properties": {
               "data": {
                 "type": "string"
               },
               "partition-key": {
                 "type": "string"
               }
             }
           }
         }
       }
     }
   }
 }
}
```

OpenAPI 2.0

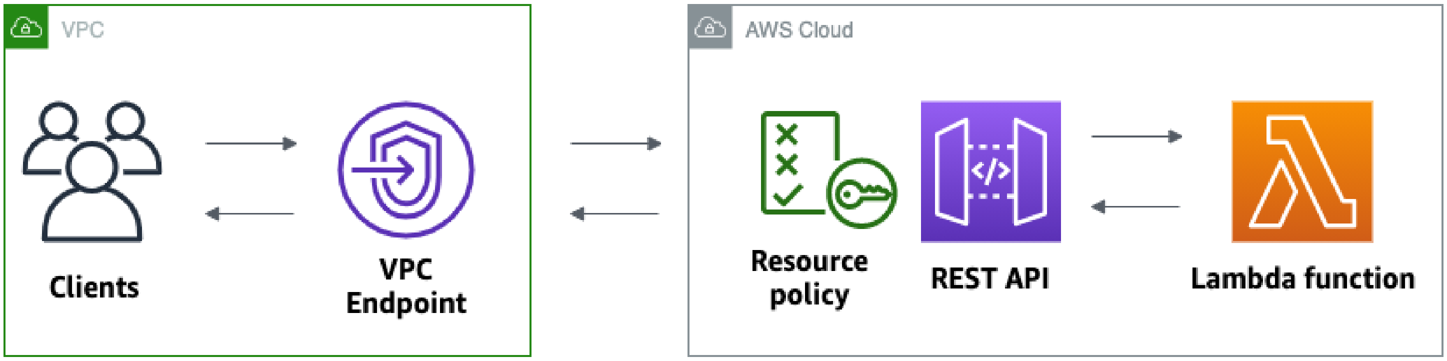
```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-03-31T18:25:32Z",
    "title": "KinesisProxy"
  },
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/streams": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/CreateStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardCount\": 5,\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "delete": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "500": {
            "description": "500 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DeleteStream",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/record": {
      "put": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecord",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Data\": \"$util.base64Encode($input.json('$.Data'))\",\n
 \"PartitionKey\": \"$input.path('$.PartitionKey')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/records": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "Shard-Iterator",
            "in": "header",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
```

```
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardIterator\": \"$input.params('Shard-
Iterator')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "put": {
        "consumes": [
          "application/json",
          "application/x-amz-json-1.1"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
          },
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "in": "body",
            "name": "PutRecordsMethodRequestPayload",
            "required": true,
            "schema": {
              "$ref": "#/definitions/PutRecordsMethodRequestPayload"
            }
          },
          {
            "in": "body",
            "name": "PutRecordsMethodRequestPayload",
            "required": true,
            "schema": {
              "$ref": "#/definitions/PutRecordsMethodRequestPayload"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Records\": [\n          {\n            \"Data\":
 \"$util.base64Encode($elem.data)\",\n            \"PartitionKey\":
 \"$elem.partition-key\"\n          }#if($foreach.hasNext),#end\n    ]\n}",
            "application/x-amz-json-1.1": "{\n  \"StreamName\":
 \"$input.params('stream-name')\",\n  \"records\" : [\n    {\n        \"Data
\" : \"$elem.data\",\n        \"PartitionKey\" : \"$elem.partition-key\"\n
  }#if($foreach.hasNext),#end\n  ]\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/sharditerator": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "shard-id",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetShardIterator",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardId\": \"$input.params('shard-
id')\",\n    \"ShardIteratorType\": \"TRIM_HORIZON\",\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object"
    },
    "PutRecordsMethodRequestPayload": {
      "type": "object",
      "properties": {
        "records": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "data": {
                "type": "string"
              },
              "partition-key": {
                "type": "string"
              }
            }
          }
        }
      }
    }
  }
}
```

Tutorial: Building a private REST API

In this tutorial, you create a private REST API. Clients can access the API only from within your Amazon VPC. The API is isolated from the public internet, which is a common security requirement.

This tutorial takes approximately 30 minutes to complete. First, you use an AWS CloudFormation template to create an Amazon VPC, a VPC endpoint, an AWS Lambda function, and launch an Amazon EC2 instance that you'll use to test your API. Next, you use the AWS Management Console to create a private API and attach a resource policy that allows access only from your VPC endpoint. Lastly, you test your API.



To complete this tutorial, you need an AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.

In this tutorial, you use the AWS Management Console. For an AWS CloudFormation template that creates this API and all related resources, see template.yaml.

Topics

- Step 1: Create dependencies
- Step 2: Create a private API
- Step 3: Create a method and integration
- Step 4: Attach a resource policy
- Step 5: Deploy your API
- Step 6: Verify that your API isn't publicly accessible
- Step 7: Connect to an instance in your VPC and invoke your API
- Step 8: Clean up
- Next steps: Automate with AWS CloudFormation

Step 1: Create dependencies

Download and unzip this AWS CloudFormation template. You use the template to create all of the dependencies for your private API, including an Amazon VPC, a VPC endpoint, and a Lambda function that serves as the backend of your API. You create the private API later.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.

3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`private-api-tutorial`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the dependencies for your API, which can take a few minutes. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, choose **Outputs**. Note your VPC endpoint ID. You need it for later steps in this tutorial.

Step 2: Create a private API

You create a private API to allow only clients within your VPC to access it.

To create a private API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**, and then for **REST API**, choose **Build**.
3. For **API name**, enter **`private-api-tutorial`**.
4. For **API endpoint type**, select **Private**.
5. For **VPC endpoint IDs**, enter the VPC endpoint ID from the **Outputs** of your AWS CloudFormation stack.
6. Choose **Create API**.

Step 3: Create a method and integration

You create a GET method and Lambda integration to handle GET requests to your API. When a client invokes your API, API Gateway sends the request to the Lambda function that you created in Step 1, and then returns a response to the client.

To create a method and integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.

2. Choose your API.
3. Select the **/** resource, and then choose **Create method**.
4. For **Method type** select GET.
5. For **Integration type**, select **Lambda function**.
6. Turn on **Lambda proxy integration**. With a Lambda proxy integration, API Gateway sends an event to Lambda with a defined structure, and transforms the response from your Lambda function to an HTTP response.
7. For **Lambda function**, choose the function that you created with the AWS CloudFormation template in Step 1. The function's name begins with **`private-api-tutorial`**.
8. Choose **Create method**.

Step 4: Attach a resource policy

You attach a resource policy to your API that allows clients to invoke your API only through your VPC endpoint. To further restrict access to your API, you can also configure a VPC endpoint policy for your VPC endpoint, but that's not necessary for this tutorial.

To attach a resource policy

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Resource policy**, and then choose **Create policy**.
4. Enter the following policy. Replace *`vpceID`* with your VPC endpoint ID from the **Outputs** of your AWS CloudFormation stack.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "execute-api:/*",
            "Condition": {
                "StringNotEquals": {
                    "aws:sourceVpce": "vpceID"
                }
            }
        },
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "execute-api:/*"
        }
    ]
}
```

5. Choose **Save changes**.

Step 5: Deploy your API

Next, you deploy your API to make it available to clients in your Amazon VPC.

To deploy an API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Deploy API**.
4. For **Stage**, select **New stage**.
5. For **Stage name**, enter **`test`**.
6. (Optional) For **Description**, enter a description.
7. Choose **Deploy**.

Now you're ready to test your API.

Step 6: Verify that your API isn't publicly accessible

Use `curl` to verify that you can't invoke your API from outside of your Amazon VPC.

To test your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. In the main navigation pane, choose **Stages**, and then choose the **test** stage.

4. Under **Stage details**, choose the copy icon to copy your API's invoke URL. The URL looks like `https://abcdef123.execute-api.us-west-2.amazonaws.com/test`. The VPC endpoint that you created in Step 1 has private DNS enabled, so you can use the provided URL to invoke your API.

5. Use curl to attempt to invoke your API from outside of your VPC.

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/test
```

   Curl indicates that your API's endpoint can't be resolved. If you get a different response, go back to Step 2, and make sure that you choose **Private** for your API's endpoint type.

```
curl: (6) Could not resolve host: abcdef123.execute-api.us-west-2.amazonaws.com/
test
```

Next, you connect to an Amazon EC2 instance in your VPC to invoke your API.

Step 7: Connect to an instance in your VPC and invoke your API

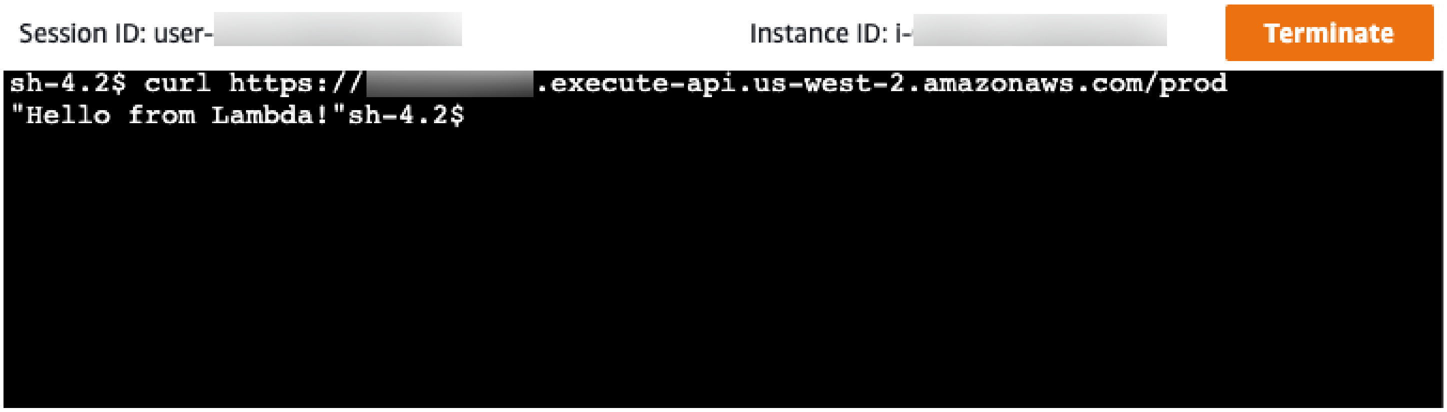
Next, you test your API from within your Amazon VPC. To access your private API, you connect to an Amazon EC2 instance in your VPC and then use curl to invoke your API. You use Systems Manager Session Manager to connect to your instance in the browser.

To test your API

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose **Instances**.
3. Choose the instance named **private-api-tutorial** that you created with the AWS CloudFormation template in Step 1.
4. Choose **Connect** and then choose **Session Manager**.
5. Choose **Connect** to launch a browser-based session to your instance.
6. In your Session Manager session, use curl to invoke your API. You can invoke your API because you're using an instance in your Amazon VPC.

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/test
```

   Verify that you get the response `Hello from Lambda!`.



```
sh-4.2$ curl https://[redacted].execute-api.us-west-2.amazonaws.com/prod
"Hello from Lambda!"sh-4.2$
```

You successfully created an API that's accessible only from within your Amazon VPC and then verified that it works.

Step 8: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this tutorial. The following steps delete your REST API and your AWS CloudFormation stack.

To delete a REST API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select an API. Choose **API actions**, choose **Delete API**, and then confirm your choice.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

Next steps: Automate with AWS CloudFormation

You can automate the creation and cleanup of all AWS resources involved in this tutorial. For a full example AWS CloudFormation template, see template.yaml.

Amazon API Gateway HTTP API tutorials

The following tutorials provide hands-on exercises to help you learn about API Gateway HTTP APIs.

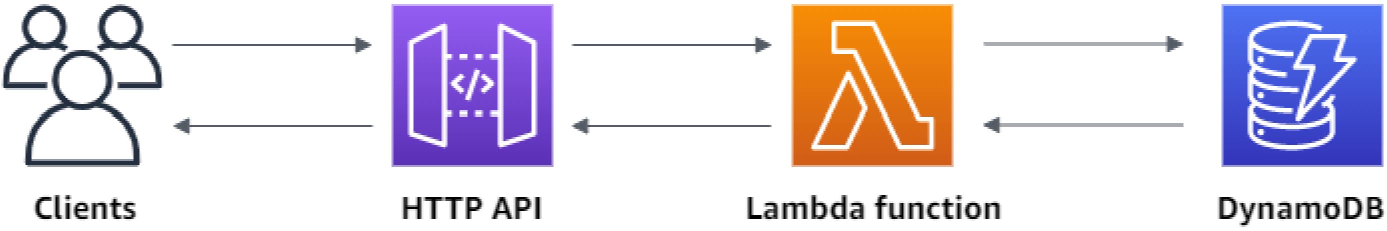
Topics

Tutorial: Build a CRUD API with Lambda and DynamoDB

In this tutorial, you create a serverless API that creates, reads, updates, and deletes items from a DynamoDB table. DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. This tutorial takes approximately 30 minutes to complete, and you can do it within the AWS Free Tier.

First, you create a DynamoDB table using the DynamoDB console. Then you create a Lambda function using the AWS Lambda console. Next, you create an HTTP API using the API Gateway console. Lastly, you test your API.

When you invoke your HTTP API, API Gateway routes the request to your Lambda function. The Lambda function interacts with DynamoDB, and returns a response to API Gateway. API Gateway then returns a response to you.



To complete this exercise, you need an AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.

In this tutorial, you use the AWS Management Console. For an AWS SAM template that creates this API and all related resources, see `template.yaml`.

Topics

Step 1: Create a DynamoDB table

You use a DynamoDB table to store data for your API.

Each item has a unique ID, which we use as the partition key for the table.

To create a DynamoDB table

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Choose **Create table**.
3. For **Table name**, enter **`http-crud-tutorial-items`**.
4. For **Partition key**, enter **`id`**.
5. Choose **Create table**.

Step 2: Create a Lambda function

You create a Lambda function for the backend of your API. This Lambda function creates, reads, updates, and deletes items from DynamoDB. The function uses events from API Gateway to determine how to interact with DynamoDB. For simplicity this tutorial uses a single Lambda function. As a best practice, you should create separate functions for each route.

To create a Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. Choose **Create function**.
3. For **Function name**, enter **`http-crud-tutorial-function`**.
4. Under **Permissions** choose **Change default execution role**.
5. Select **Create a new role from AWS policy templates**.
6. For **Role name**, enter **`http-crud-tutorial-role`**.

7. For **Policy templates**, choose **`Simple microservice permissions`**. This policy grants the Lambda function permission to interact with DynamoDB.

   > **Note**
   >
   > This tutorial uses a managed policy for simplicity. As a best practice, you should create your own IAM policy to grant the minimum permissions required.

8. Choose **Create function**.

9. Open `index.mjs` in the console's code editor, and replace its contents with the following code. Choose **Deploy** to update your function.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ScanCommand,
  PutCommand,
  GetCommand,
  DeleteCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});

const dynamo = DynamoDBDocumentClient.from(client);

const tableName = "http-crud-tutorial-items";

export const handler = async (event, context) => {
  let body;
  let statusCode = 200;
  const headers = {
    "Content-Type": "application/json",
  };

  try {
    switch (event.routeKey) {
      case "DELETE /items/{id}":
        await dynamo.send(
          new DeleteCommand({
            TableName: tableName,
            Key: {
              id: event.pathParameters.id,
            },
          })
        );
        body = `Deleted item ${event.pathParameters.id}`;
        break;
      case "GET /items/{id}":
        body = await dynamo.send(
          new GetCommand({
            TableName: tableName,
            Key: {
              id: event.pathParameters.id,
            },
          })
        );
        body = body.Item;
        break;
      case "GET /items":
        body = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        body = body.Items;
        break;
      case "PUT /items":
        let requestJSON = JSON.parse(event.body);
        await dynamo.send(
          new PutCommand({
            TableName: tableName,
            Item: {
              id: requestJSON.id,
              price: requestJSON.price,
              name: requestJSON.name,
            },
          })
        );
        body = `Put item ${requestJSON.id}`;
        break;
      default:
        throw new Error(`Unsupported route: "${event.routeKey}"`);
    }
  } catch (err) {
    statusCode = 400;
    body = err.message;
  } finally {
    body = JSON.stringify(body);
  }

  return {
    statusCode,
    body,
    headers,
  };
};
```

Step 3: Create an HTTP API

The HTTP API provides an HTTP endpoint for your Lambda function. In this step, you create an empty API. In the following steps, you configure routes and integrations to connect your API and your Lambda function.

To create an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**, and then for **HTTP API**, choose **Build**.
3. For **API name**, enter **`http-crud-tutorial-api`**.
4. Choose **Next**.
5. For **Configure routes**, choose **Next** to skip route creation. You create routes later.
6. Review the stage that API Gateway creates for you, and then choose **Next**.
7. Choose **Create**.

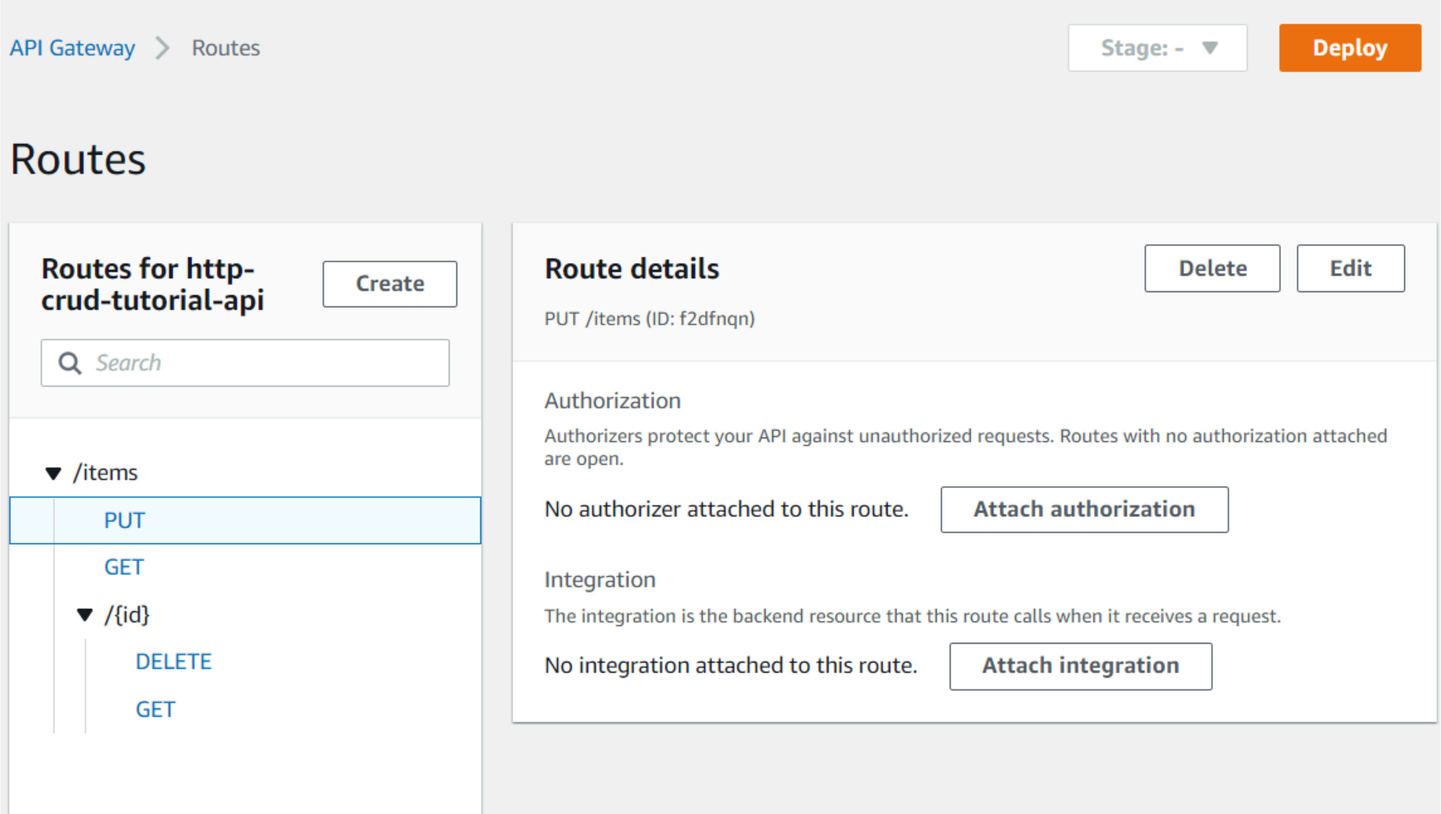
Step 4: Create routes

Routes are a way to send incoming API requests to backend resources. Routes consist of two parts: an HTTP method and a resource path, for example, `GET /items`. For this example API, we create four routes:

- `GET /items/{id}`
- `GET /items`
- `PUT /items`
- `DELETE /items/{id}`

To create routes

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Routes**.
4. Choose **Create**.
5. For **Method**, choose **`GET`**.
6. For the path, enter **`/items/{id}`**. The `{id}` at the end of the path is a path parameter that API Gateway retrieves from the request path when a client makes a request.
7. Choose **Create**.
8. Repeat steps 4-7 for `GET /items`, `DELETE /items/{id}`, and `PUT /items`.



PUT /items (ID: f2dfnqn)

Authorizers protect your API against unauthorized requests. Routes with no authorization attached are open.

No authorizer attached to this route.

The integration is the backend resource that this route calls when it receives a request.

No integration attached to this route.

Step 5: Create an integration

You create an integration to connect a route to backend resources. For this example API, you create one Lambda integration that you use for all routes.

To create an integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Integrations**.
4. Choose **Manage integrations** and then choose **Create**.
5. Skip **Attach this integration to a route**. You complete that in a later step.
6. For **Integration type**, choose **Lambda function**.
7. For **Lambda function**, enter **`http-crud-tutorial-function`**.
8. Choose **Create**.

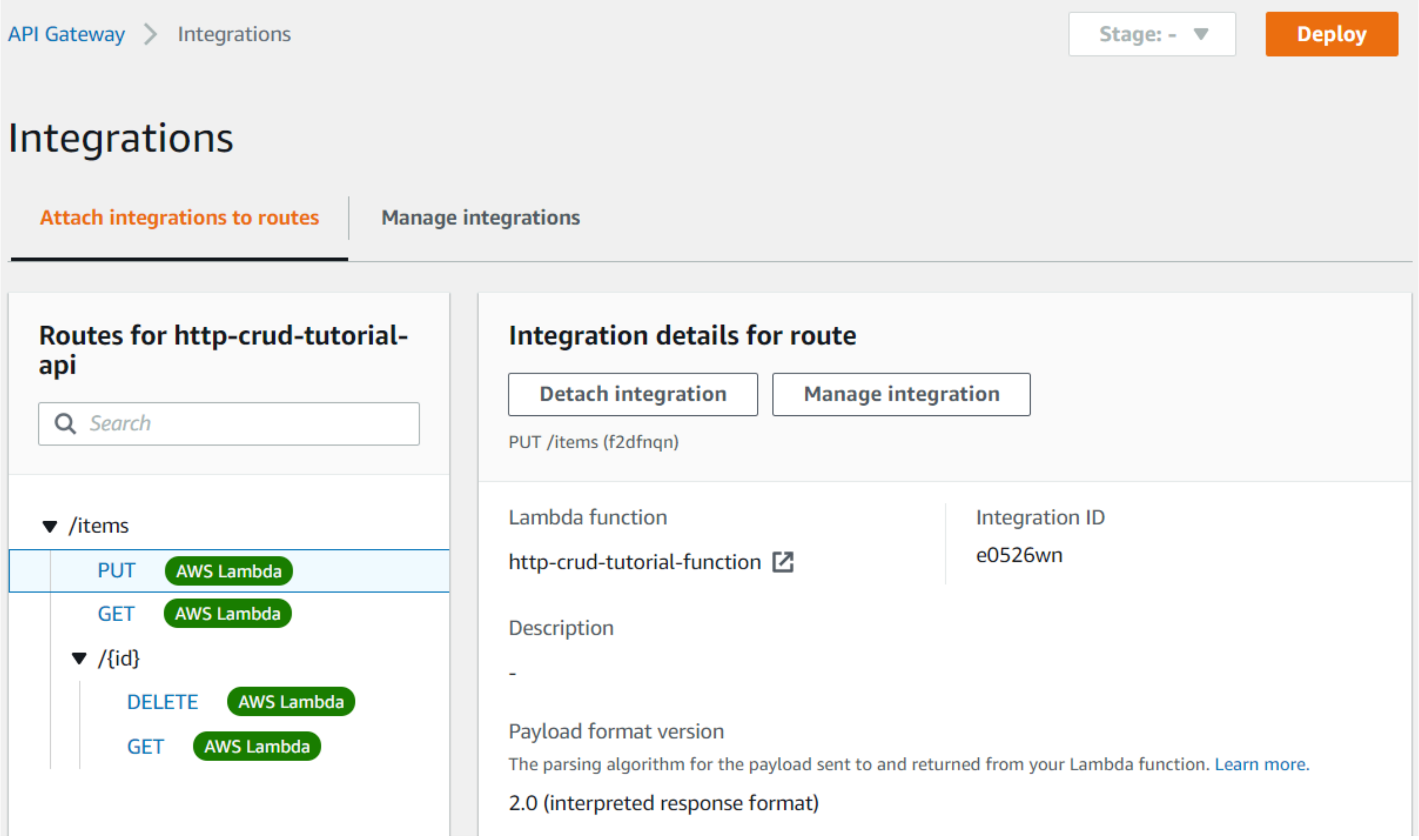
Step 6: Attach your integration to routes

For this example API, you use the same Lambda integration for all routes. After you attach the integration to all of the API's routes, your Lambda function is invoked when a client calls any of your routes.

To attach integrations to routes

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Integrations**.
4. Choose a route.
5. Under **Choose an existing integration**, choose **`http-crud-tutorial-function`**.
6. Choose **Attach integration**.
7. Repeat steps 4-6 for all routes.

All routes show that an AWS Lambda integration is attached.



Routes for http-crud-tutorial-api

/items
PUT AWS Lambda
GET AWS Lambda
/{id}
DELETE AWS Lambda
GET AWS Lambda

Integration details for route

PUT /items (f2dfnqn)

Lambda function
http-crud-tutorial-function

Integration ID
e0526wn

Description
-

Payload format version
The parsing algorithm for the payload sent to and returned from your Lambda function. Learn more.
2.0 (interpreted response format)

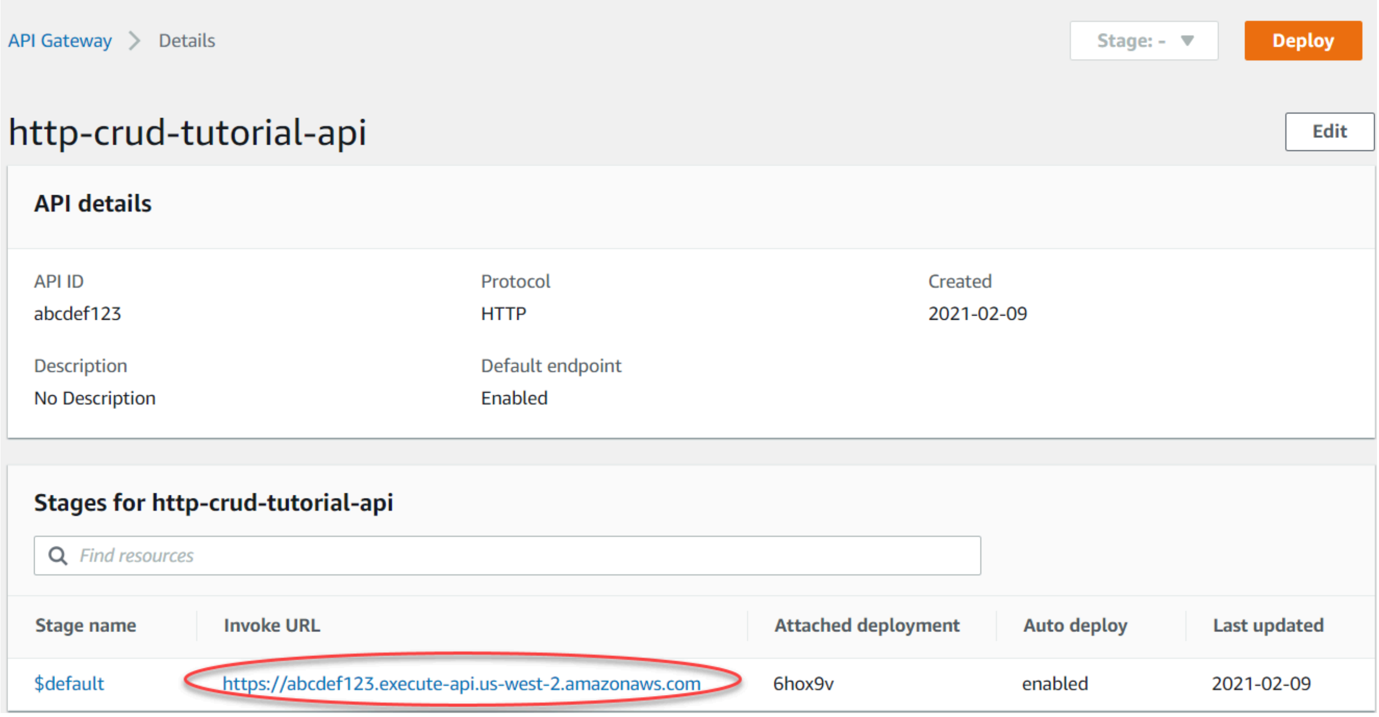
Now that you have an HTTP API with routes and integrations, you can test your API.

Step 7: Test your API

To make sure that your API is working, you use curl.

To get the URL to invoke your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Note your API's invoke URL. It appears under **Invoke URL** on the **Details** page.



http-crud-tutorial-api

API details

API ID abcdef123	Protocol HTTP	Created 2021-02-09
Description No Description	Default endpoint Enabled	

Stages for http-crud-tutorial-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-west-2.amazonaws.com	6hox9v	enabled	2021-02-09

4. Copy your API's invoke URL.

   The full URL looks like `https://abcdef123.execute-api.us-west-2.amazonaws.com`.

To create or update an item

- Use the following command to create or update an item. The command includes a request body with the item's ID, price, and name.

```
curl -X "PUT" -H "Content-Type: application/json" -d "{\"id\": \"123\",
 \"price\": 12345, \"name\": \"myitem\"}" https://abcdef123.execute-api.us-
west-2.amazonaws.com/items
```

To get all items

- Use the following command to list all items.

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items
```

To get an item

- Use the following command to get an item by its ID.

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items/123
```

To delete an item

1. Use the following command to delete an item.

```
curl -X "DELETE" https://abcdef123.execute-api.us-west-2.amazonaws.com/items/123
```

2. Get all items to verify that the item was deleted.

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items
```

Step 8: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this getting started exercise. The following steps delete your HTTP API, your Lambda function, and associated resources.

To delete a DynamoDB table

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Select your table.
3. Choose **Delete table**.
4. Confirm your choice, and choose **Delete**.

To delete an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select an API. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function

1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. On the **Functions** page, select a function. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function's log group

1. In the Amazon CloudWatch console, open the Log groups page.
2. On the **Log groups** page, select the function's log group (`/aws/lambda/http-crud-tutorial-function`). Choose **Actions**, and then choose **Delete log group**.
3. Choose **Delete**.

To delete a Lambda function's execution role

1. In the AWS Identity and Access Management console, open the Roles page.
2. Select the function's role, for example, `http-crud-tutorial-role`.
3. Choose **Delete role**.
4. Choose **Yes, delete**.

Next steps: Automate with AWS SAM or AWS CloudFormation

You can automate the creation and cleanup of AWS resources by using AWS CloudFormation or AWS SAM. For an example AWS SAM template for this tutorial, see `template.yaml`.

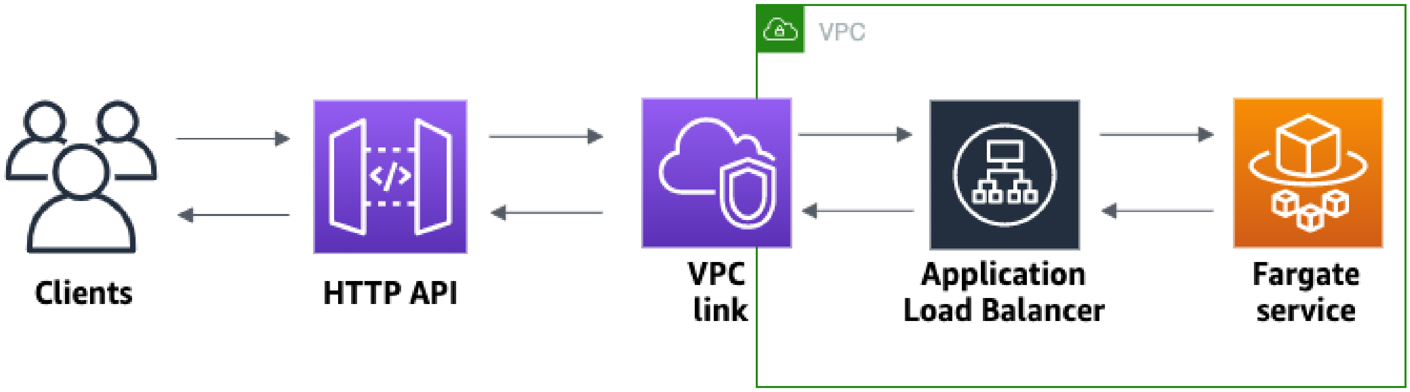
For example AWS CloudFormation templates, see example AWS CloudFormation templates.

Tutorial: Building an HTTP API with a private integration to an Amazon ECS service

In this tutorial, you create a serverless API that connects to an Amazon ECS service that runs in an Amazon VPC. Clients outside of your Amazon VPC can use the API to access your Amazon ECS service.

This tutorial takes approximately an hour to complete. First, you use an AWS CloudFormation template to create a Amazon VPC and Amazon ECS service. Then you use the API Gateway console to create a VPC link. The VPC link allows API Gateway to access the Amazon ECS service that runs in your Amazon VPC. Next, you create an HTTP API that uses the VPC link to connect to your Amazon ECS service. Lastly, you test your API.

When you invoke your HTTP API, API Gateway routes the request to your Amazon ECS service through your VPC link, and then returns the response from the service.



To complete this tutorial, you need an AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.

In this tutorial, you use the AWS Management Console. For an AWS CloudFormation template that creates this API and all related resources, see template.yaml.

Topics

Step 1: Create an Amazon ECS service

Amazon ECS is a container management service that makes it easy to run, stop, and manage Docker containers on a cluster. In this tutorial, you run your cluster on a serverless infrastructure that's managed by Amazon ECS.

Download and unzip this AWS CloudFormation template, which creates all of the dependencies for the service, including an Amazon VPC. You use the template to create an Amazon ECS service that uses an Application Load Balancer.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.
3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`http-api-private-integrations-tutorial`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the ECS service, which can take a few minutes. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, you're ready to move on to the next step.

Step 2: Create a VPC link

A VPC link allows API Gateway to access private resources in an Amazon VPC. You use a VPC link to allow clients to access your Amazon ECS service through your HTTP API.

To create a VPC link

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the main navigation pane, choose **VPC links** and then choose **Create**.

   You might need to choose the menu icon to open the main navigation pane.
3. For **Choose a VPC link version**, select **VPC link for HTTP APIs**.
4. For **Name**, enter **`private-integrations-tutorial`**.
5. For **VPC**, choose the VPC that you created in step 1. The name should start with **PrivateIntegrationsStack**.

6. For **Subnets**, select the two private subnets in your VPC. Their names end with `PrivateSubnet`.
7. Choose **Create**.

After you create your VPC link, API Gateway provisions Elastic Network Interfaces to access your VPC. The process can take a few minutes. In the meantime, you can create your API.

Step 3: Create an HTTP API

The HTTP API provides an HTTP endpoint for your Amazon ECS service. In this step, you create an empty API. In Steps 4 and 5, you configure a route and an integration to connect your API and your Amazon ECS service.

To create an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**, and then for **HTTP API**, choose **Build**.
3. For **API name**, enter **`http-private-integrations-tutorial`**.
4. Choose **Next**.
5. For **Configure routes**, choose **Next** to skip route creation. You create routes later.
6. Review the stage that API Gateway creates for you. API Gateway creates a `$default` stage with automatic deployments enabled, which is the best choice for this tutorial. Choose **Next**.
7. Choose **Create**.

Step 4: Create a route

Routes are a way to send incoming API requests to backend resources. Routes consist of two parts: an HTTP method and a resource path, for example, `GET /items`. For this example API, we create one route.

To create a route

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Routes**.

4. Choose **Create**.
5. For **Method**, choose **`ANY`**.
6. For the path, enter **`/{proxy+}`**. The `{proxy+}` at the end of the path is a greedy path variable. API Gateway sends all requests to your API to this route.
7. Choose **Create**.

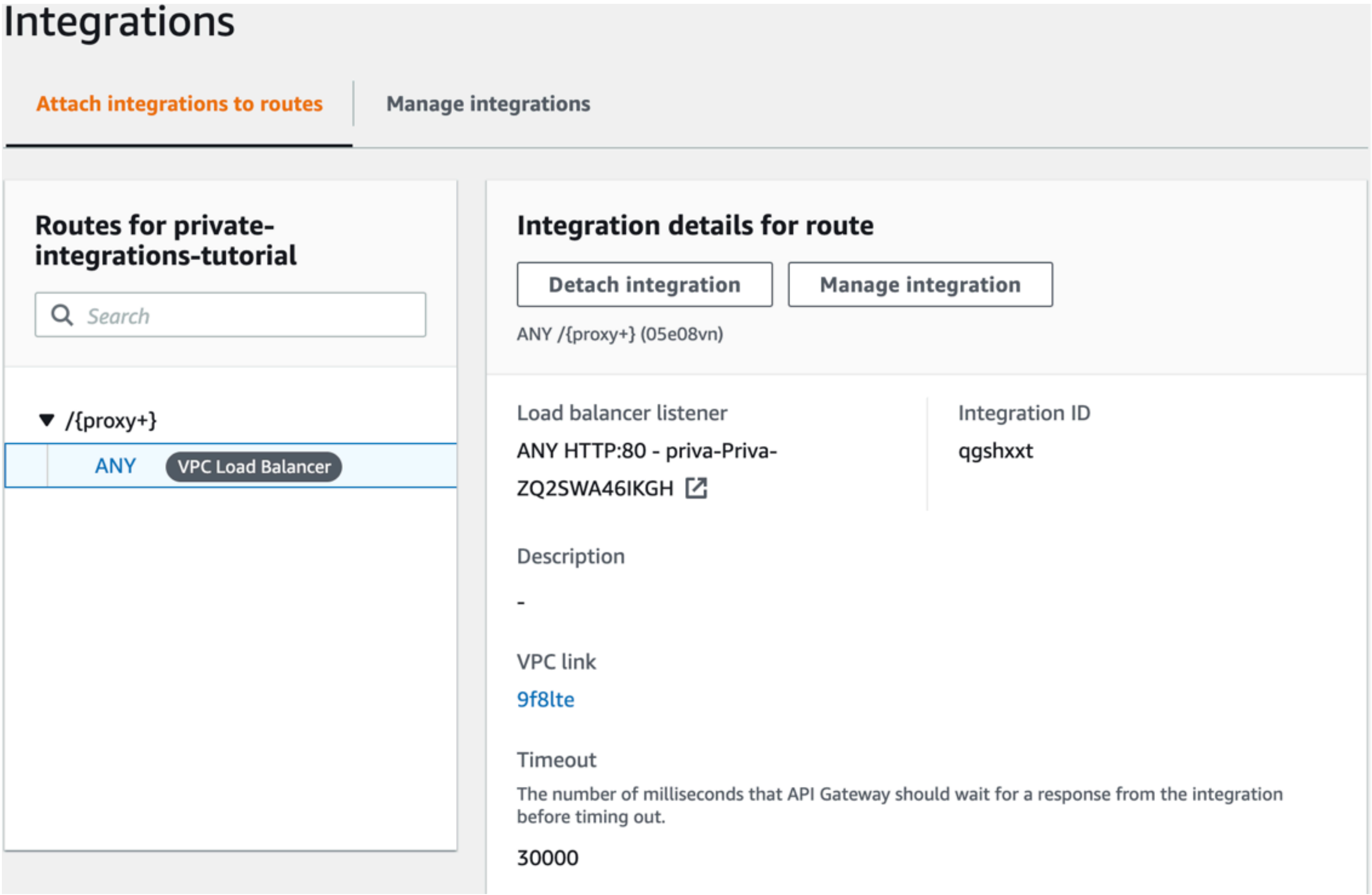
Step 5: Create an integration

You create an integration to connect a route to backend resources.

To create an integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Integrations**.
4. Choose **Manage integrations** and then choose **Create**.
5. For **Attach this integration to a route**, select the **ANY /{proxy+}** route that you created earlier.
6. For **Integration type**, choose **Private resource**.
7. For **Integration details**, choose **Select manually**.
8. For **Target service**, choose **ALB/NLB**.
9. For **Load balancer**, choose the load balancer that you created with the AWS CloudFormation template in Step 1. It's name should start with **http-Priva**.
10. For **Listener**, choose **`HTTP 80`**.
11. For **VPC link**, choose the VPC link that you created in Step 2. It's name should be `private-integrations-tutorial`.
12. Choose **Create**.

To verify that your route and integration are set up correctly, select **Attach integrations to routes**. The console shows that you have an `ANY /{proxy+}` route with an integration to a VPC Load Balancer.



ANY /{proxy+} (05e08vn)

Load balancer listener
ANY HTTP:80 - priva-Priva-ZQ2SWA46IKGH

Integration ID
qgshxxt

Description
-

VPC link
9f8lte

Timeout
The number of milliseconds that API Gateway should wait for a response from the integration before timing out.
30000

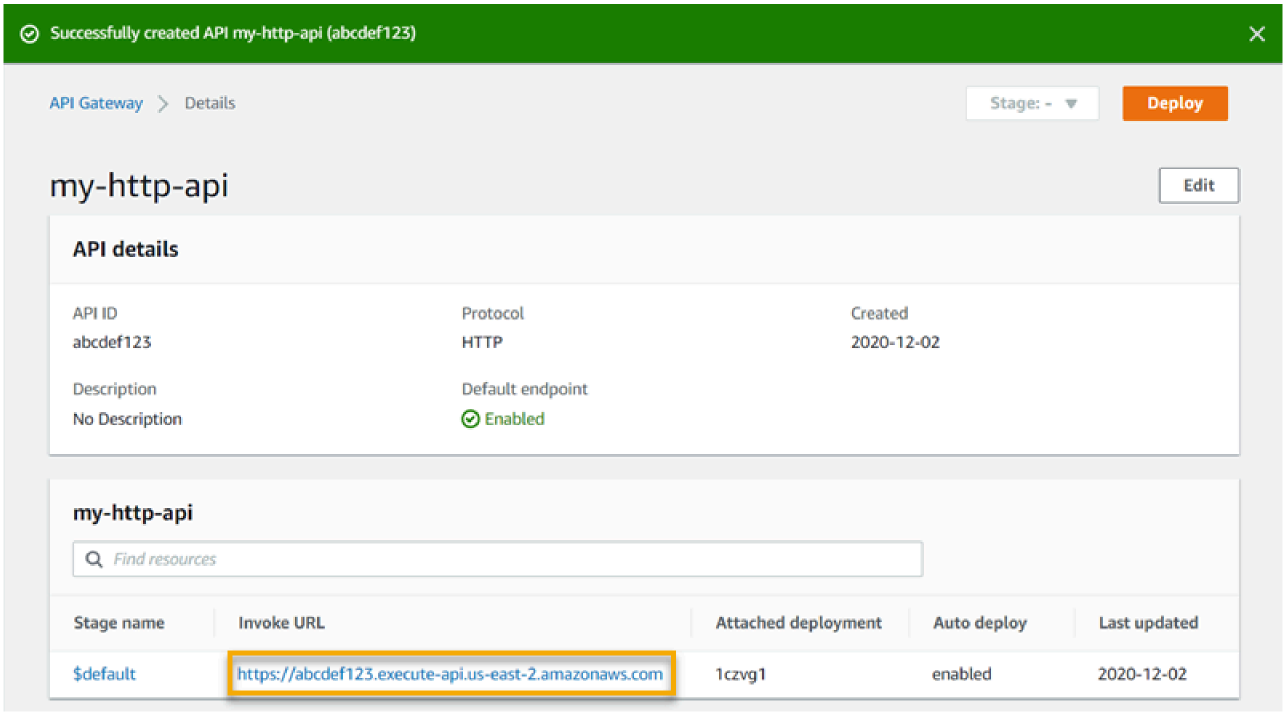
Now you're ready to test your API.

Step 6: Test your API

Next, you test your API to make sure that it's working. For simplicity, use a web browser to invoke your API.

To test your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Note your API's invoke URL.



Successfully created API my-http-api (abcdef123)

my-http-api

API details

API ID
abcdef123

Protocol
HTTP

Created
2020-12-02

Description
No Description

Default endpoint
✓ Enabled

my-http-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-east-2.amazonaws.com	1czvg1	enabled	2020-12-02

4. In a web browser, go to your API's invoke URL.

   The full URL should look like `https://`*`abcdef123`*`.execute-api.`*`us-east-2`*`.amazonaws.com`.

   Your browser sends a GET request to the API.

5. Verify that your API's response is a welcome message that tells you that your app is running on Amazon ECS.

   If you see the welcome message, you successfully created an Amazon ECS service that runs in an Amazon VPC, and you used an API Gateway HTTP API with a VPC link to access the Amazon ECS service.

Step 7: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this tutorial. The following steps delete your VPC link, AWS CloudFormation stack, and HTTP API.

To delete an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select an API. Choose **Actions**, choose **Delete**, and then confirm your choice.

To delete a VPC link

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **VPC link**.
3. Select your VPC link, choose **Delete**, and then confirm your choice.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

Next steps: Automate with AWS CloudFormation

You can automate the creation and cleanup of all AWS resources involved in this tutorial. For a full example AWS CloudFormation template, see template.yaml.

Amazon API Gateway WebSocket API tutorials

The following tutorials provide a hands-on exercise to help you learn about API Gateway WebSocket APIs.

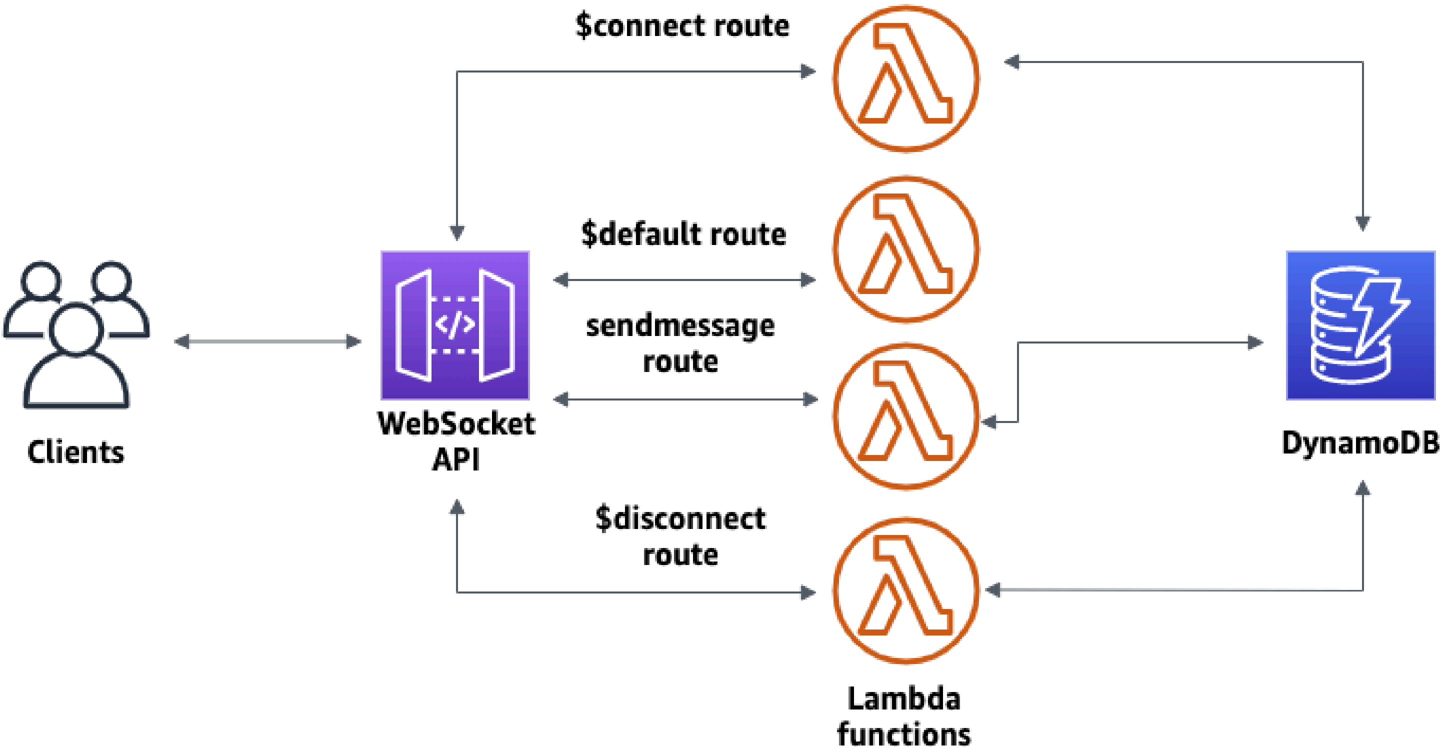
Topics

- Tutorial: Building a serverless chat app with a WebSocket API, Lambda and DynamoDB
- Tutorial: Building a serverless application with three integration types

Tutorial: Building a serverless chat app with a WebSocket API, Lambda and DynamoDB

In this tutorial, you'll create a serverless chat application with a WebSocket API. With a WebSocket API, you can support two-way communication between clients. Clients can receive messages without having to poll for updates.

This tutorial takes approximately 30 minutes to complete. First, you'll use an AWS CloudFormation template to create Lambda functions that will handle API requests, as well as a DynamoDB table that stores your client IDs. Then, you'll use the API Gateway console to create a WebSocket API that integrates with your Lambda functions. Lastly, you'll test your API to verify that messages are sent and received.



To complete this tutorial, you need an AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.

You also need `wscat` to connect to your API. For more information, see the section called "Use `wscat` to connect to a WebSocket API and send messages to it".

Topics

- Step 1: Create Lambda functions and a DynamoDB table
- Step 2: Create a WebSocket API
- Step 3: Test your API
- Step 4: Clean up
- Next steps: Automate with AWS CloudFormation

Step 1: Create Lambda functions and a DynamoDB table

Download and unzip the app creation template for AWS CloudFormation. You'll use this template to create a Amazon DynamoDB table to store your app's client IDs. Each connected client has a unique ID which we will use as the table's partition key. This template also creates Lambda functions that update your client connections in DynamoDB and handle sending messages to connected clients.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.
3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`websocket-api-chat-app-tutorial`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, you're ready to move on to the next step.

Step 2: Create a WebSocket API

You'll create a WebSocket API to handle client connections and route requests to the Lambda functions that you created in Step 1.

To create a WebSocket API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**. Then for **WebSocket API**, choose **Build**.
3. For **API name**, enter **`websocket-chat-app-tutorial`**.
4. For **Route selection expression**, enter **`request.body.action`**. The route selection expression determines the route that API Gateway invokes when a client sends a message.
5. Choose **Next**.
6. For **Predefined routes**, choose **Add $connect**, **Add $disconnect**, and **Add $default**. The **$connect** and **$disconnect** routes are special routes that API Gateway invokes automatically

when a client connects to or disconnects from an API. API Gateway invokes the `$default` route when no other routes match a request.

7. For **Custom routes**, choose **Add custom route**. For **Route key**, enter **`sendmessage`**. This custom route handles messages that are sent to connected clients.
8. Choose **Next**.
9. Under **Attach integrations**, for each route and **Integration type**, choose Lambda.

   For **Lambda**, choose the corresponding Lambda function that you created with AWS CloudFormation in Step 1. Each function's name matches a route. For example, for the **$connect** route, choose the function named **`websocket-chat-app-tutorial-ConnectHandler`**.
10. Review the stage that API Gateway creates for you. By default, API Gateway creates a stage name `production` and automatically deploys your API to that stage. Choose **Next**.
11. Choose **Create and deploy**.

Step 3: Test your API

Next, you'll test your API to make sure that it works correctly. Use the `wscat` command to connect to the API.

To to get the invoke URL for your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Choose **Stages**, and then choose **production**.
4. Note your API's **WebSocket URL**. The URL should look like `wss://*abcdef123*.execute-api.*us-east-2*.amazonaws.com/production`.

To connect to your API

1. Use the following command to connect to your API. When you connect to your API, API Gateway invokes the `$connect` route. When this route is invoked, it calls a Lambda function that stores your connection ID in DynamoDB.

   ```
   wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
   ```

```
Connected (press CTRL+C to quit)
```

2. Open a new terminal and run the **wscat** command again with the following parameters.

```
wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
```

```
Connected (press CTRL+C to quit)
```

This gives you two connected clients that can exchange messages.

To send a message

- API Gateway determines which route to invoke based on your API's route selection expression. Your API's route selection expression is `$request.body.action`. As a result, API Gateway invokes the `sendmessage` route when you send the following message:

```
{"action": "sendmessage", "message": "hello, everyone!"}
```

The Lambda function associated with the invoked route collects the client IDs from DynamoDB. Then, the function calls the API Gateway Management API and sends the message to those clients. All connected clients receive the following message:

```
< hello, everyone!
```

To invoke your API's $default route

- API Gateway invokes your API's default route when a client sends a message that doesn't match your defined routes. The Lambda function associated with the `$default` route uses the API Gateway Management API to send the client information about their connection.

```
test
```

```
Use the sendmessage route to send a message. Your info:
 {"ConnectedAt":"2022-01-25T18:50:04.673Z","Identity":
{"SourceIp":"192.0.2.1","UserAgent":null},"LastActiveAt":"2022-01-25T18:50:07.642Z","connec
```

To disconnect from your API

- Press **CTRL+C** to disconnect from your API. When a client disconnects from your API, API Gateway invokes your API's `$disconnect` route. The Lambda integration for your API's `$disconnect` route removes the connection ID from DynamoDB.

Step 4: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this tutorial. The following steps delete your AWS CloudFormation stack and WebSocket API.

To delete a WebSocket API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select your `websocket-chat-app-tutorial` API. Choose **Actions**, choose **Delete**, and then confirm your choice.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

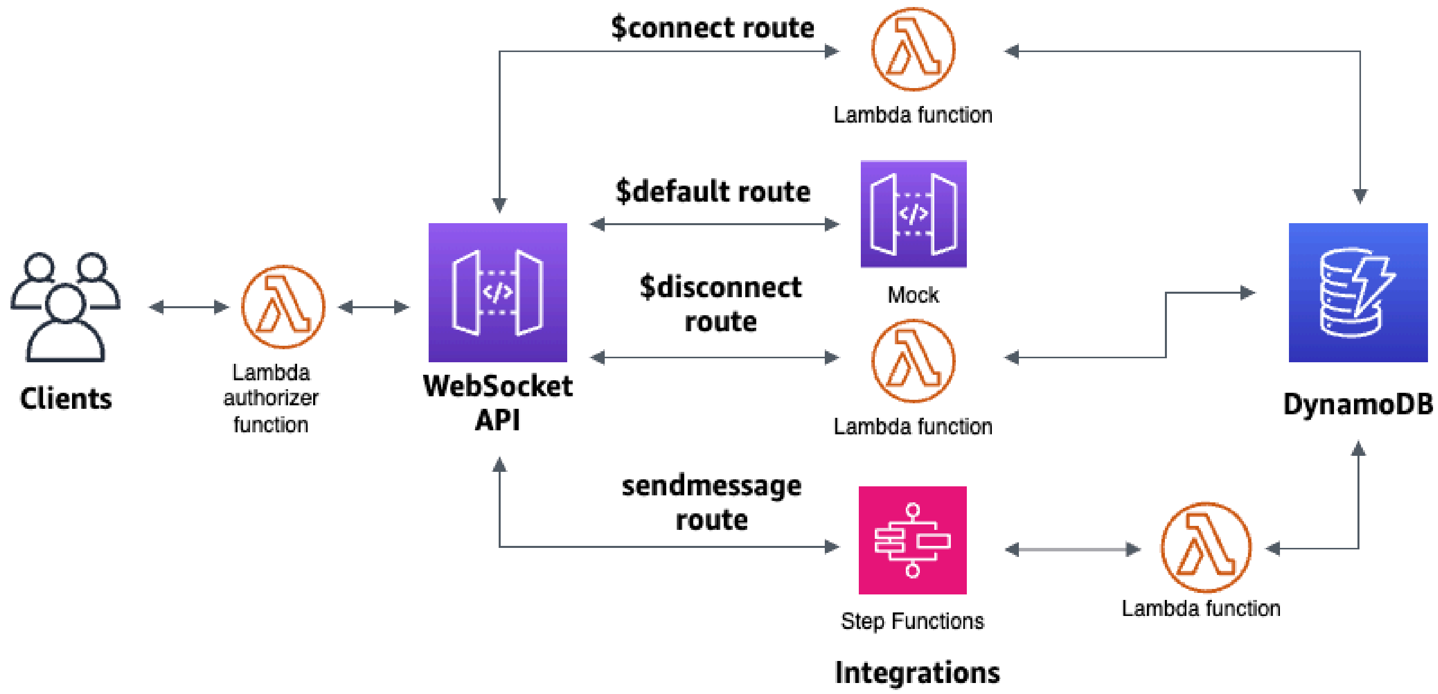
Next steps: Automate with AWS CloudFormation

You can automate the creation and cleanup of all of the AWS resources involved in this tutorial. For an AWS CloudFormation template that creates this API and all related resources, see chat-app.yaml.

Tutorial: Building a serverless application with three integration types

In this tutorial, you create a serverless broadcast application with a WebSocket API. Clients can receive messages without having to poll for updates.

This tutorial shows how to broadcast messages to connected clients and includes an example of a Lambda authorizer, a mock integration, and a non-proxy integration to Step Functions.



After you create your resources using a AWS CloudFormation template, you'll use the API Gateway console to create a WebSocket API that integrates with your AWS resources. You'll attach a Lambda authorizer to your API and create an AWS service integration with Step Functions to start a state machine execution. The Step Functions state machine will invoke a Lambda function that sends a message to all connected clients.

After you build your API, you'll test your connection to your API and verify that messages are sent and received. This tutorial takes approximately 45 minutes to complete.

Topics

Prerequisites

You need the following prerequisites:

- An AWS account and an AWS Identity and Access Management user with console access. For more information, see *Prerequisites*.
- `wscat` to connect to your API. For more information, see the section called "Use `wscat` to connect to a WebSocket API and send messages to it".

We recommend that you complete the WebSocket chat app tutorial before you start this tutorial. To complete the WebSocket chat app tutorial, see the section called "WebSocket chat app".

Step 1: Create resources

Download and unzip the app creation template for AWS CloudFormation. You'll use this template to create the following:

- Lambda functions that handle API requests and authorize access to your API.
- A DynamoDB table to store client IDs.
- A Step Functions state machine to send messages to connected clients.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.
3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`websocket-step-functions-tutorial`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. Choose the **Outputs** tab to see your created resources and their ARNs. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, you're ready to move on to the next step.

Step 2: Create a WebSocket API

You'll create a WebSocket API to handle client connections and route requests to the resources that you created in Step 1.

To create a WebSocket API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**. Then for **WebSocket API**, choose **Build**.
3. For **API name**, enter **`websocket-step-functions-tutorial`**.
4. For **Route selection expression**, enter **`request.body.action`**.

   The route selection expression determines the route that API Gateway invokes when a client sends a message.
5. Choose **Next**.
6. For **Predefined routes**, choose **Add $connect**, **Add $disconnect**, **Add $default**.

   The **$connect** and **$disconnect** routes are special routes that API Gateway invokes automatically when a client connects to or disconnects from an API. API Gateway invokes the **$default** route when no other routes match a request. You will create a custom route to connect to Step Functions after you create your API.
7. Choose **Next**.
8. For **Integration for $connect**, do the following:

   a. For **Integration type**, choose **Lambda**.

   b. For **Lambda function**, choose the corresponding **$connect** Lambda function that you created with AWS CloudFormation in Step 1. The Lambda function name should start with **`websocket-step`**.
9. For **Integration for $disconnect**, do the following:

   a. For **Integration type**, choose **Lambda**.

   b. For **Lambda function**, choose the corresponding **$disconnect** Lambda function that you created with AWS CloudFormation in Step 1. The Lambda function name should start with **`websocket-step`**.

10. For **Integration for $default**, choose **mock**.

    In a mock integration, API Gateway manages the route response without an integration backend.

11. Choose **Next**.
12. Review the stage that API Gateway creates for you. By default, API Gateway creates a stage named **production** and automatically deploys your API to that stage. Choose **Next**.
13. Choose **Create and deploy**.

Step 3: Create a Lambda authorizer

To control access to your WebSocket API, you create a Lambda authorizer. The AWS CloudFormation template created the Lambda authorizer function for you. You can see the Lambda function in the Lambda console. The name should start with **`websocket-step-functions-tutorial-AuthorizerHandler`**. This Lambda function denies all calls to the WebSocket API unless the `Authorization` header is `Allow`.

In this step, you configure the **$connect** route to use the Lambda authorizer.

To create a Lambda authorizer

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the main navigation pane, choose **Authorizers**.
3. Choose **Create an authorizer**.
4. For **Authorizer name**, enter **`LambdaAuthorizer`**.
5. For **Authorizer ARN**, enter the name of the authorizer created by the AWS CloudFormation template. The name should start with **`websocket-step-functions-tutorial-AuthorizerHandler`**.

   > **Note**
   >
   > We recommend that you don't use this example authorizer for your production APIs.

6. For **Identity source type**, choose **Header**. For **Key**, enter **`Authorization`**.
7. Choose **Create authorizer**.

After you create your authorizer, you attach it to the **$connect** route of your API.

To attach an authorizer to the $connect route

1. In the main navigation pane, choose **Routes**.
2. Choose the **$connect** route.
3. In the **Route request settings** section, choose **Edit**.
4. For **Authorization**, choose the dropdown menu, and then select your request authorizer.
5. Choose **Save changes**.

Step 4: Create a mock two-way integration

Next, you create the two-way mock integration for the **$default** route. A mock integration lets you send a response to the client without using a backend. When you create an integration for the **$default** route, you can show clients how to interact with your API.

You configure the **$default** route to inform clients to use the **sendmessage** route.

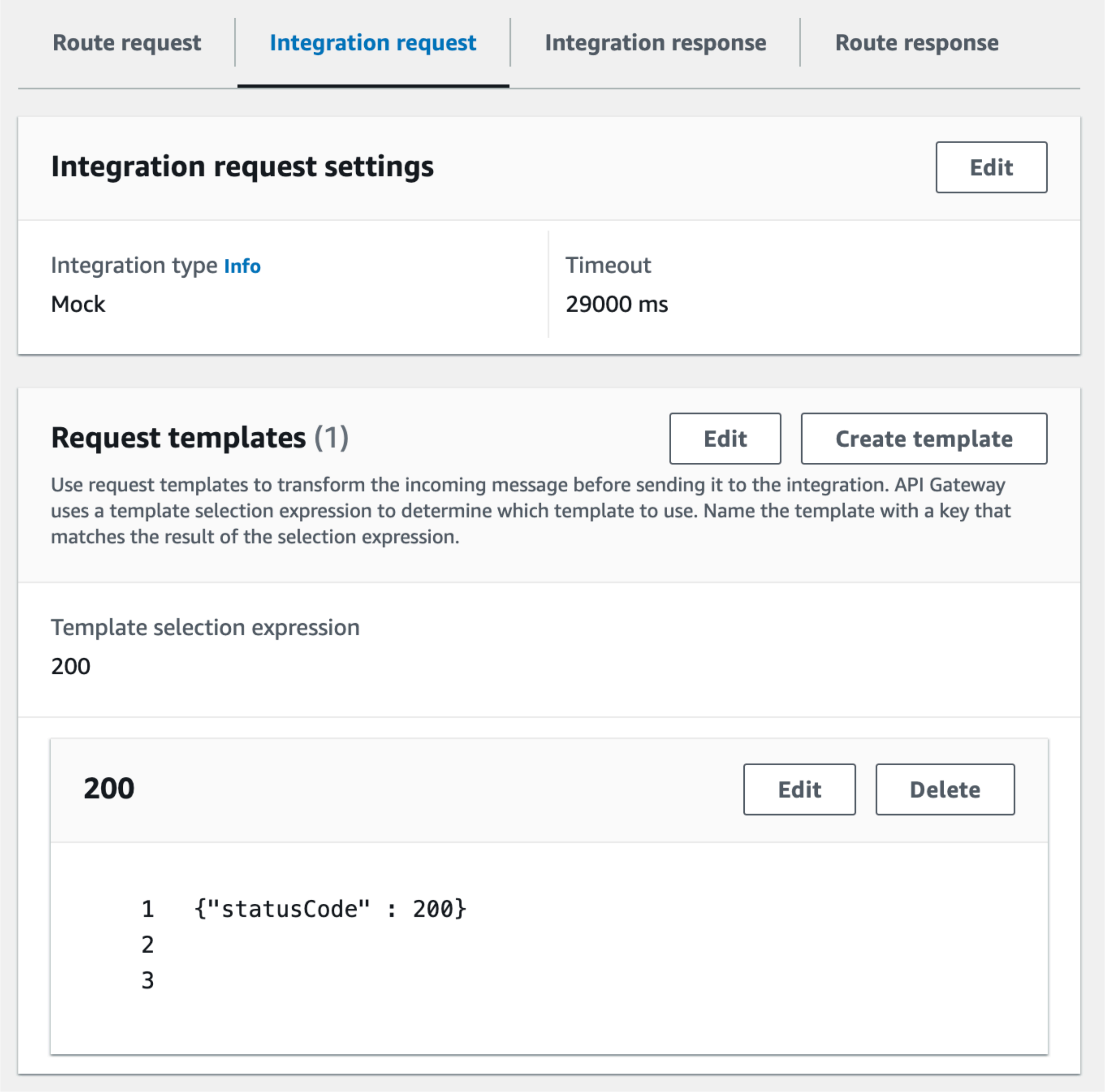
To create a mock integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose the **$default** route, and then choose the **Integration request** tab.
3. For **Request templates**, choose **Edit**.
4. For **Template selection expression**, enter **`200`**, and then choose **Edit**.
5. On the **Integration request** tab, for **Request templates**, choose **Create template**.
6. For **Template key**, enter **`200`**.
7. For **Generate template**, enter the following mapping template:

   ```
   {"statusCode": 200}
   ```

   Choose **Create template**.

   The result should look like the following:



Integration request settings

Integration type
Mock

Timeout
29000 ms

Request templates (1)

Use request templates to transform the incoming message before sending it to the integration. API Gateway uses a template selection expression to determine which template to use. Name the template with a key that matches the result of the selection expression.

Template selection expression
200

200

```
{"statusCode" : 200}


```

8. The the **$default route** pane, choose **Enable two-way communication**.
9. Choose the **Integration response** tab, and then choose **Create integration response**.
10. For **Response key**, enter **`$default`**.
11. For **Template selection expression**, enter **`200`**.
12. Choose **Create response**.
13. Under **Response templates**, choose **Create template**.

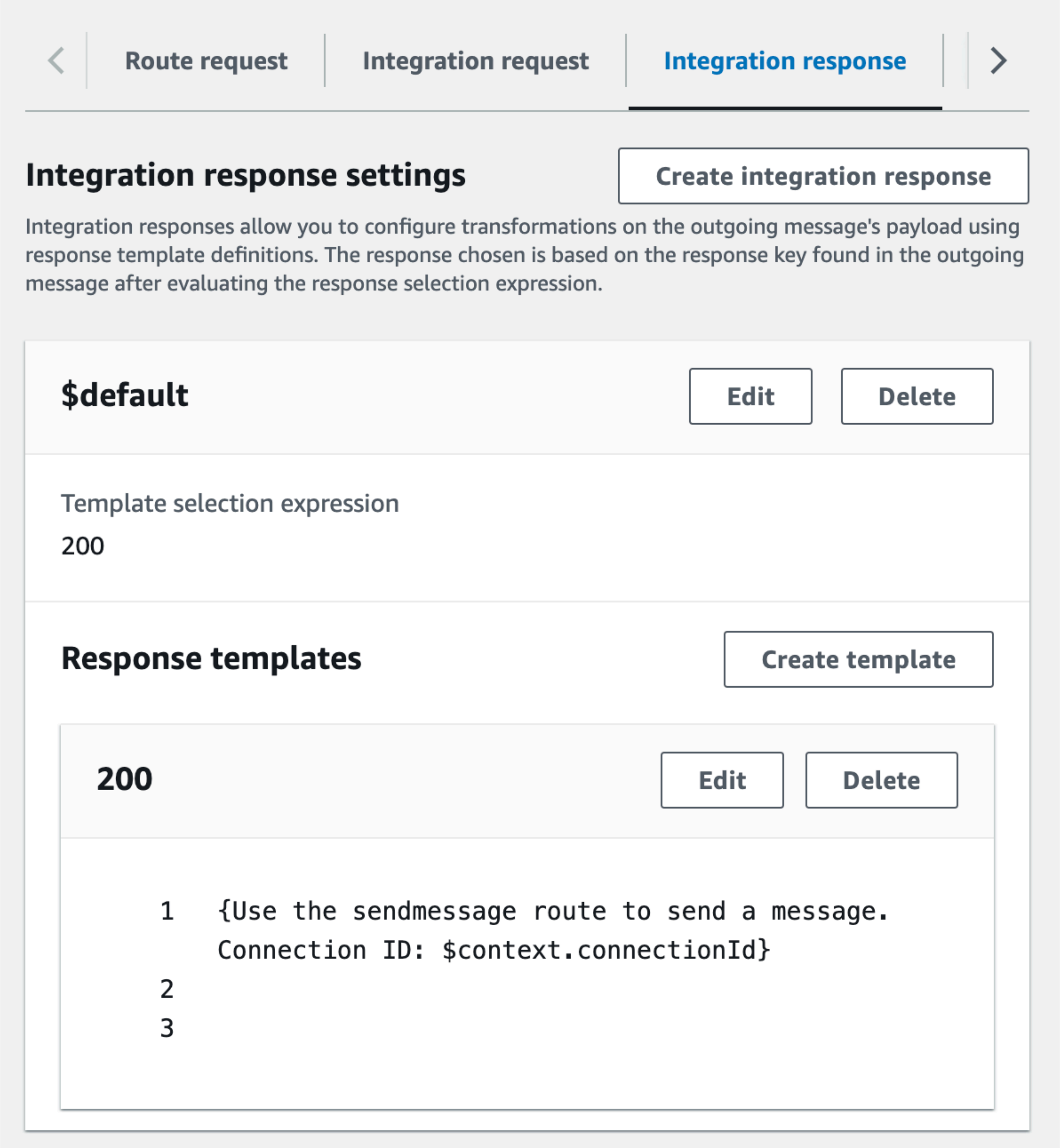
14. For **Template key**, enter **`200`**.

15. For **Response template**, enter the following mapping template:

    ```
    {"Use the sendmessage route to send a message. Connection ID:
     $context.connectionId"}
    ```

16. Choose **Create template**.

    The result should look like the following:



Integration response settings

Integration responses allow you to configure transformations on the outgoing message's payload using response template definitions. The response chosen is based on the response key found in the outgoing message after evaluating the response selection expression.

$default

Template selection expression

200

Response templates

200

```
{Use the sendmessage route to send a message. Connection ID: $context.connectionId}


```

Step 5: Create a non-proxy integration with Step Functions

Next, you create a **sendmessage** route. Clients can invoke the **sendmessage** route to broadcast a message to all connected clients. The **sendmessage** route has a non-proxy AWS service integration with AWS Step Functions. The integration invokes the StartExecution command for the Step Functions state machine that the AWS CloudFormation template created for you.

To create a non-proxy integration

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create route**.
3. For **Route key**, enter **`sendmessage`**.
4. For **Integration type**, choose **AWS service**.
5. For **AWS Region**, enter the Region where you deployed your AWS CloudFormation template.
6. For **AWS service**, choose **Step Functions**.
7. For **HTTP method**, choose **POST**.
8. For **Action name**, enter **`StartExecution`**.
9. For **Execution role**, enter the execution role created by the AWS CloudFormation template. The name should be **WebsocketTutorialApiRole**.
10. Choose **Create route**.

Next, you create a mapping template to send request parameters to the Step Functions state machine.

To create a mapping template

1. Choose the **sendmessage** route, and then choose the **Integration request** tab.
2. In the **Request templates** section, choose **Edit**.
3. For **Template selection expression**, enter **`\$default`**.
4. Choose **Edit**.
5. In the **Request templates** section, choose **Create template**.
6. For **Template key**, enter **`\$default`**.
7. For **Generate template**, enter the following mapping template:

```
#set($domain = "$context.domainName")
#set($stage = "$context.stage")
#set($body = $input.json('$'))
#set($getMessage = $util.parseJson($body))
#set($mymessage = $getMessage.message)
```

```
{
"input": "{\"domain\": \"$domain\", \"stage\": \"$stage\", \"message\": \"$mymessage\"}",
"stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:WebSocket-Tutorial-StateMachine"
}
```

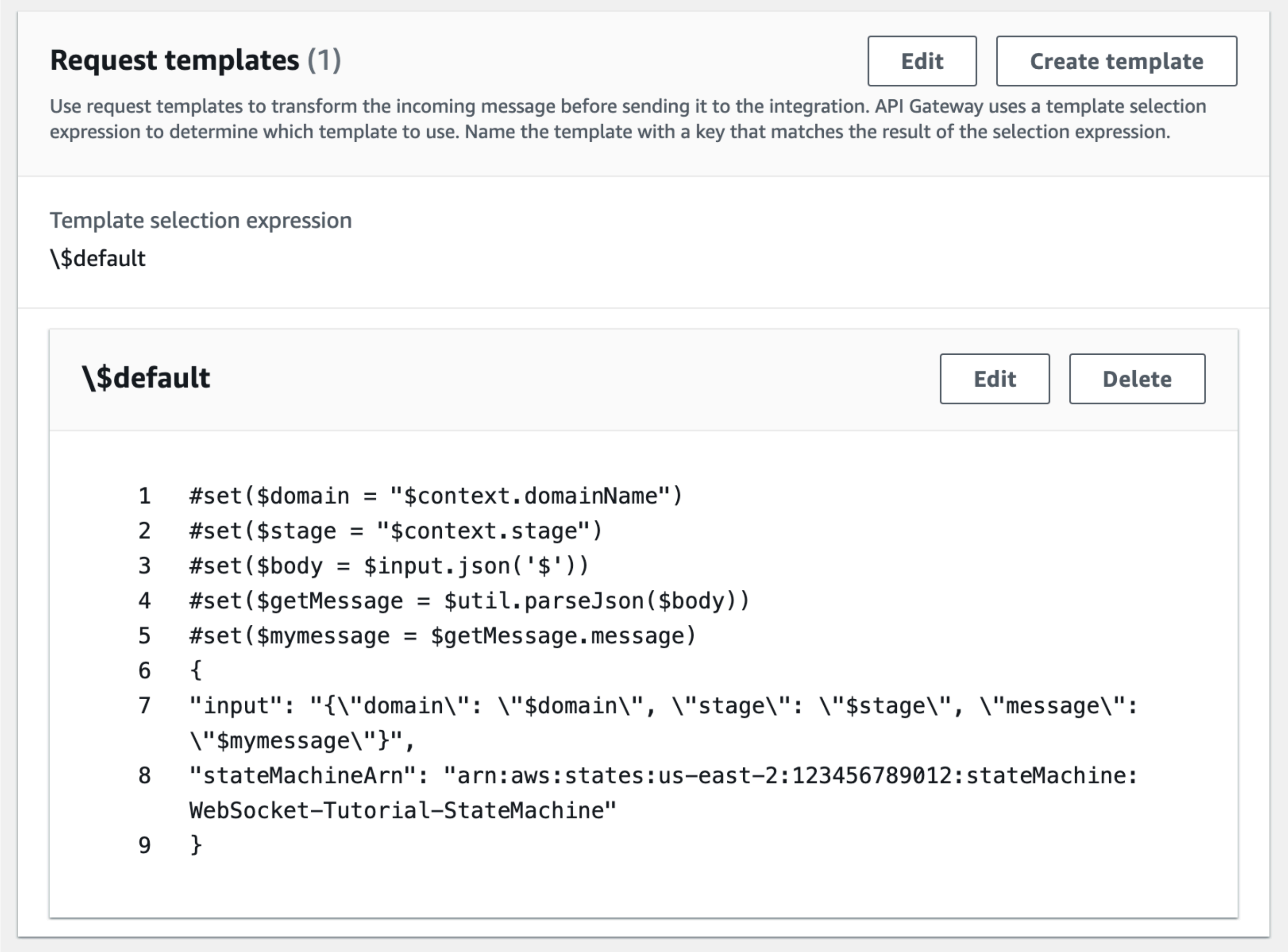
Replace the *`stateMachineArn`* with the ARN of the state machine created by AWS CloudFormation.

The mapping template does the following:

- Creates the variable `$domain` using the context variable `domainName`.
- Creates the variable `$stage` using the context variable `stage`.

  The `$domain` and `$stage` variables are required to build a callback URL.
- Takes in the incoming `sendmessage` JSON message, and extracts the `message` property.
- Creates the input for the state machine. The input is the domain and stage of the WebSocket API and the message from the `sendmessage` route.

8. Choose **Create template**.



Request templates (1)

Use request templates to transform the incoming message before sending it to the integration. API Gateway uses a template selection expression to determine which template to use. Name the template with a key that matches the result of the selection expression.

Template selection expression

\$default

\$default

```
#set($domain = "$context.domainName")
#set($stage = "$context.stage")
#set($body = $input.json('$'))
#set($getMessage = $util.parseJson($body))
#set($mymessage = $getMessage.message)
{
"input": "{\"domain\": \"$domain\", \"stage\": \"$stage\", \"message\": \"$mymessage\"}",
"stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:WebSocket-Tutorial-StateMachine"
}
```

Step 6: Test your API

Next, you'll deploy and test your API to make sure that it works correctly. You will use the `wscat` command to connect to the API and then, you will use a slash command to send a ping frame to check the connection to the WebSocket API.

To deploy your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the main navigation pane, choose **Routes**.
3. Choose **Deploy API**.
4. For **Stage**, choose **production**.
5. (Optional) For **Deployment description**, enter a description.

6. Choose **Deploy**.

After you deploy your API, you can invoke it. Use the invoke URL to call your API.

To get the invoke URL for your API

1. Choose your API.
2. Choose **Stages**, and then choose **production**.
3. Note your API's **WebSocket URL**. The URL should look like `wss://abcdef123.execute-api.us-east-2.amazonaws.com/production`.

Now that you have your invoke URL, you can test the connection to your WebSocket API.

To test the connection to your API

1. Use the following command to connect to your API. First, you test the connection by invoking the `/ping` path.

   ```
   wscat -c wss://abcdef123.execute-api.us-east-2.amazonaws.com/production -H
    "Authorization: Allow" --slash -P
   ```

   ```
   Connected (press CTRL+C to quit)
   ```

2. Enter the following command to ping the control frame. You can use a control frame for keepalive purposes from the client side.

   ```
   /ping
   ```

   The result should look like the following:

   ```
   < Received pong (data: "")
   ```

Now that you have tested the connection, you can test that your API works correctly. In this step, you open a new terminal window so the WebSocket API can send a message to all connected clients.

To test your API

1. Open a new terminal and run the `wscat` command again with the following parameters.

   ```
   wscat -c wss://abcdef123.execute-api.us-east-2.amazonaws.com/production -H
    "Authorization: Allow"
   ```

   ```
   Connected (press CTRL+C to quit)
   ```

2. API Gateway determines which route to invoke based on your API's route request selection expression. Your API's route select expression is `$request.body.action`. As a result, API Gateway invokes the `sendmessage` route when you send the following message:

   ```
   {"action": "sendmessage", "message": "hello, from Step Functions!"}
   ```

   The Step Functions state machine associated with the route invokes a Lambda function with the message and the callback URL. The Lambda function calls the API Gateway Management API and sends the message to all connected clients. All clients receive the following message:

   ```
   < hello, from Step Functions!
   ```

Now that you have tested your WebSocket API, you can disconnect from your API.

To disconnect from your API

- Press CTRL+Cto disconnect from your API.

  When a client disconnects from your API, API Gateway invokes your API's **$disconnect** route. The Lambda integration for your API's **$disconnect** route removes the connection ID from DynamoDB.

Step 7: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this tutorial. The following steps delete your AWS CloudFormation stack and WebSocket API.

To delete a WebSocket API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the **APIs** page, select your **websocket-api**.
3. Choose **Actions**, choose **Delete**, and then confirm your choice.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

Next steps

You can automate the creation and cleanup of all the AWS resources involved in this tutorial. For an example of an AWS CloudFormation template that automates these actions for this tutorial, see ws-sfn.zip.

Working with REST APIs

A REST API in API Gateway is a collection of resources and methods that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. You can use API Gateway features to help you with all aspects of the API lifecycle, from creation through monitoring your production APIs.

API Gateway REST APIs use a request/response model where a client sends a request to a service and the service responds back synchronously. This kind of model is suitable for many different kinds of applications that depend on synchronous communication.

Topics

Developing a REST API in API Gateway

This section provides details about API Gateway capabilities that you need while you're developing your API Gateway APIs.

As you're developing your API Gateway API, you decide on a number of characteristics of your API. These characteristics depend on the use case of your API. For example, you might want to only allow certain clients to call your API, or you might want it to be available to everyone. You might want an API call to execute a Lambda function, make a database query, or call an application.

Topics

Creating a REST API in Amazon API Gateway

In Amazon API Gateway, you build a REST API as a collection of programmable entities known as API Gateway resources. For example, you use a `RestApi` resource to represent an API that can contain a collection of `Resource` entities. Each `Resource` entity can in turn have one or more `Method` resources. Expressed in the request parameters and body, a `Method` defines the application programming interface for the client to access the exposed `Resource` and represents an incoming request submitted by the client. You then create an `Integration` resource to integrate the `Method` with a backend endpoint, also known as the integration endpoint, by forwarding the incoming request to a specified integration endpoint URI. If necessary, you transform request parameters or body to meet the backend requirements. For responses, you can create a `MethodResponse` resource to represent a request response received by the client and you create an `IntegrationResponse` resource to represent the request response that is returned by the backend. You can configure the integration response to transform the backend response data before returning the data to the client or to pass the backend response as-is to the client.

To help your customers understand your API, you can also provide documentation for the API, as part of the API creation or after the API is created. To enable this, add a `DocumentationPart` resource for a supported API entity.

To control how clients call an API, use IAM permissions, a Lambda authorizer, or an Amazon Cognito user pool. To meter the use of your API, set up usage plans to throttle API requests. You can enable these when creating or updating the API.

You can perform these and other tasks by using the API Gateway console, the API Gateway REST API, the AWS CLI, or one of the AWS SDKs. We discuss how to perform these tasks next.

Topics

- Set up REST API methods in API Gateway

Choose an endpoint type to set up for an API Gateway API

An *API endpoint* type refers to the hostname of the API. The API endpoint type can be *edge-optimized*, *regional*, or *private*, depending on where the majority of your API traffic originates from.

Edge-optimized API endpoints

An *edge-optimized API endpoint* typically routes requests to the nearest CloudFront Point of Presence (POP), which could help in cases where your clients are geographically distributed. This is the default endpoint type for API Gateway REST APIs.

Edge-optimized APIs capitalize the names of HTTP headers (for example, `Cookie`).

CloudFront sorts HTTP cookies in natural order by cookie name before forwarding the request to your origin. For more information about the way CloudFront processes cookies, see Caching Content Based on Cookies.

Any custom domain name that you use for an edge-optimized API applies across all regions.

Regional API endpoints

A *regional API endpoint* is intended for clients in the same region. When a client running on an EC2 instance calls an API in the same region, or when an API is intended to serve a small number of clients with high demands, a regional API reduces connection overhead.

For a regional API, any custom domain name that you use is specific to the region where the API is deployed. If you deploy a regional API in multiple regions, it can have the same custom domain name in all regions. You can use custom domains together with Amazon Route 53 to perform tasks such as latency-based routing. For more information, see the section called “Setting up a regional custom domain name” and the section called “Creating an edge-optimized custom domain name”.

Regional API endpoints pass all header names through as-is.

Private API endpoints

A *private API endpoint* is an API endpoint that can only be accessed from your Amazon Virtual Private Cloud (VPC) using an interface VPC endpoint, which is an endpoint network interface (ENI) that you create in your VPC. For more information, see the section called “Private APIs”.

Private API endpoints pass all header names through as-is.

Change a public or private API endpoint type in API Gateway

Changing an API endpoint type requires you to update the API's configuration. You can change an existing API type using the API Gateway console, the AWS CLI, or an AWS SDK for API Gateway. The endpoint type cannot be changed again until the current change is completed, but your API will be available.

The following endpoint type changes are supported:

- From edge-optimized to regional or private
- From regional to edge-optimized or private
- From private to regional

You cannot change a private API into an edge-optimized API.

If you are changing a public API from edge-optimized to regional or vice versa, note that an edge-optimized API may have different behaviors than a regional API. For example, an edge-optimized API removes the `Content-MD5` header. Any MD5 hash value passed to the backend can be expressed in a request string parameter or a body property. However, the regional API passes this header through, although it may remap the header name to some other name. Understanding the differences helps you decide how to update an edge-optimized API to a regional one or from a regional API to an edge-optimized one.

Topics

Use the API Gateway console to change an API endpoint type

To change the API endpoint type of your API, perform one of the following sets of steps:

To convert a public endpoint from Regional or edge-optimized and vice versa

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.

3. Choose **API settings**.
4. In the **API details** section, choose **Edit**.
5. For **API endpoint type**, select either **Edge-optimized** or **Regional**.
6. Choose **Save changes**.
7. Redeploy your API so that the changes will take effect.

To convert a private endpoint to a regional endpoint

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Edit the resource policy for your API to remove any mention of VPCs or VPC endpoints so that API calls from outside your VPC as well as inside your VPC will succeed.
4. Choose **API settings**.
5. In the **API details** section, choose **Edit**.
6. For **API endpoint type**, select **Regional**.
7. Choose **Save changes** to start the update.
8. Remove the resource policy from your API.
9. Redeploy your API so that the changes will take effect.

Use the AWS CLI to change an API endpoint type

To use the AWS CLI to update an edge-optimized API whose API ID is *{api-id}*, call update-rest-api as follows:

```
aws apigateway update-rest-api \
    --rest-api-id {api-id} \
    --patch-operations op=replace,path=/endpointConfiguration/types/EDGE,value=REGIONAL
```

The successful response has a status code of `200 OK` and a payload similar to the following:

```
{

    "createdDate": "2017-10-16T04:09:31Z",
    "description": "Your first API with Amazon API Gateway. This is a sample API that
 integrates via HTTP with our demo Pet Store endpoints",
    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "id": "0gsnjtjck8",
    "name": "PetStore imported as edge-optimized"
}
```

Conversely, update a regional API to an edge-optimized API as follows:

```
aws apigateway update-rest-api \
    --rest-api-id {api-id} \
    --patch-operations op=replace,path=/endpointConfiguration/types/REGIONAL,value=EDGE
```

Because put-rest-api is for updating API definitions, it is not applicable to updating an API endpoint type.

Initialize REST API setup in API Gateway

You can create a REST API using the API Gateway console, the API Gateway REST API, the AWS SDKs for API Gateway, and the AWS Command Line Interface.

When you create a REST API using the API Gateway REST API, the AWS SDKs for API Gateway, or the AWS Command Line Interface, the default configuration is an edge-optimized API. For more information about API endpoint types, see the section called “Choose an API endpoint type”.

When you deploy your API to a stage, your API Gateway creates a default URL for your API. For more information about the default URL, see the section called “Deploying a REST API”. You can assign a custom domain name (for example, `apis.example.com`) as the API's host name and call the API with a base URL of the `https://apis.example.com/myApi` format. For more information about custom domain names, see the section called “Custom domain names”.

We recommend that you use one of the following examples to learn how to create a REST API.

Topics

- Set up a Regional API in API Gateway

Set up an API using the API Gateway console

We recommend that you choose from the following tutorials to learn how to create a REST API Gateway using the REST API Gateway console.

To create a REST API that passes an event to a Lambda function, choose the section called "Getting started with the REST API console".

To create a REST API where you configure the integration request payload to a Lambda function, choose the section called "Tutorial: Build an API with Lambda non-proxy integration".

To create a REST API that has an integration with an HTTP endpoint, choose the section called "Tutorial: Build a REST API with HTTP proxy integration".

To create a REST API that where you configure the integration request to an HTTP endpoint, choose the section called "Tutorial: Build an API with HTTP non-proxy integration".

To import an example API, choose the section called "Tutorial: Create a REST API by importing an example".

Alternatively, you can set up an API by using the API Gateway Import API feature to upload an external API definition, such as one expressed in OpenAPI 2.0 with the Working with API Gateway extensions to OpenAPI. The example provided in Tutorial: Create a REST API by importing an example uses the Import API feature.

Set up an edge-optimized API using AWS CLI commands

Setting up an API using the AWS CLI requires working with the `create-rest-api`, `create-resource` or `get-resources`, `put-method`, `put-method-response`, `put-integration`, and `put-integration-response` commands. The following procedures show how to work with these AWS CLI commands to create the simple PetStore API of the HTTP integration type.

To create a simple PetStore API using AWS CLI

1. Call the `create-rest-api` command to set up the `RestApi` in a specific region (`us-west-2`).

   ```
   aws apigateway create-rest-api --name 'Simple PetStore (AWS CLI)' --region us-
   west-2
   ```

The following is the output of this command:

```
{
    "id": "vaz7da96z6",
    "name": "Simple PetStore (AWS CLI)",
    "createdDate": "2022-12-15T08:07:04-08:00",
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "EDGE"
        ]
    },
    "disableExecuteApiEndpoint": false
}
```

Note the returned `id` of the newly created `RestApi`. You need it to set up other parts of the API.

2. Call the `get-resources` command to retrieve the root resource identifier of the `RestApi`.

```
aws apigateway get-resources --rest-api-id vaz7da96z6 --region us-west-2
```

The following is the output of this command:

```
{
    "items": [
        {
            "id": "begaltmsm8",
            "path": "/"
        }
    ]
}
```

Note the root resource `Id`. You need it to start setting the API's resource tree and configuring methods and integrations.

3. Call the `create-resource` command to append a child resource (`pets`) under the root resource (`begaltmsm8`):

```
aws apigateway create-resource --rest-api-id vaz7da96z6 \
      --region us-west-2 \
      --parent-id begaltmsm8 \
      --path-part pets
```

The following is the output of this command:

```
{
    "id": "6sxz2j",
    "parentId": "begaltmsm8",
    "pathPart": "pets",
    "path": "/pets"
}
```

To append a child resource under the root, you specify the root resource `Id` as the `parentId` property value. Similarly, to append a child resource under the `pets` resource, you repeat the preceding step while replacing the `parent-id` value with the `pets` resource `id` of `6sxz2j`:

```
aws apigateway create-resource --rest-api-id vaz7da96z6 \
      --region us-west-2 \
      --parent-id 6sxz2j \
      --path-part '{petId}'
```

To make a path part a path parameter, enclose it in a pair of curly brackets. If successful, this command returns the following response:

```
{
    "id": "rjkmth",
    "parentId": "6sxz2j",
    "path": "/pets/{petId}",
    "pathPart": "{petId}"
}
```

Now that you created two resources: `/pets` (`6sxz2j`) and `/pets/{petId}` (`rjkmth`), you can proceed to set up methods on them.

4. Call the `put-method` command to add the `GET` HTTP method on the `/pets` resource. This creates an API `Method` of `GET /pets` with open access, referencing the `/pets` resource by its ID value of `6sxz2j`.

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2
```

The following is the successful output of this command:

```
{
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "apiKeyRequired": false
}
```

The method is for open access because `authorization-type` is set to `NONE`. To permit only authenticated users to call the method, you can use IAM roles and policies, a Lambda authorizer (formerly known as a custom authorizer), or an Amazon Cognito user pool. For more information, see the section called “Access control”.

To enable read access to the `/pets/{petId}` resource (`rjkmth`), add the `GET` HTTP method on it to create an API `Method` of `GET /pets/{petId}` as follows.

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id rjkmth --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-parameters method.request.path.petId=true
```

The following is the successful output of this command:

```
{
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "apiKeyRequired": false,
    "requestParameters": {
        "method.request.path.petId": true
    }
}
```

Note that the method request path parameter of `petId` must be specified as a required request parameter for its dynamically set value to be mapped to a corresponding integration request parameter and passed to the backend.

5. Call the `put-method-response` command to set up the 200 OK response of the `GET /pets` method, specifying the `/pets` resource by its ID value of `6sxz2j`.

   ```
   aws apigateway put-method-response --rest-api-id vaz7da96z6 \
          --resource-id 6sxz2j --http-method GET \
          --status-code 200  --region us-west-2
   ```

   The following is the output of this command:

   ```
   {
       "statusCode": "200"
   }
   ```

   Similarly, to set the 200 OK response of the `GET /pets/{petId}` method, do the following, specifying the `/pets/{petId}` resource by its resource ID value of `rjkmth`:

   ```
   aws apigateway put-method-response --rest-api-id vaz7da96z6 \
          --resource-id rjkmth --http-method GET \
          --status-code 200  --region us-west-2
   ```

   Having set up a simple client interface for the API, you can proceed to set up the integration of the API methods with the backend.

6. Call the `put-integration` command to set up an `Integration` with a specified HTTP endpoint for the `GET /pets` method. The `/pets` resource is identified by its resource Id `6sxz2j`:

   ```
   aws apigateway put-integration --rest-api-id vaz7da96z6 \
          --resource-id 6sxz2j --http-method GET --type HTTP \
          --integration-http-method GET \
          --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets' \
          --region us-west-2
   ```

   The following is the output of this command:

   ```
   {
       "type": "HTTP",
       "httpMethod": "GET",
       "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
       "connectionType": "INTERNET",
   ```

```
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "6sxz2j",
    "cacheKeyParameters": []
}
```

Notice that the integration `uri` of `http://petstore-demo-endpoint.execute-api.com/petstore/pets` specifies the integration endpoint of the `GET /pets` method.

Similarly, you create an integration request for the `GET /pets/{petId}` method as follows:

```
aws apigateway put-integration \
        --rest-api-id vaz7da96z6 \
        --resource-id rjkmth \
        --http-method GET \
        --type HTTP \
        --integration-http-method GET \
        --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}' \
        --request-parameters
 '{"integration.request.path.id":"method.request.path.petId"}' \
        --region us-west-2
```

Here, the integration endpoint, `uri` of `http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`, also uses a path parameter (`id`). Its value is mapped from the corresponding method request path parameter of `{petId}`. The mapping is defined as part of the `request-parameters`. If this mapping is not defined here, the client gets an error response when trying to call the method.

The following is the output of this command:

```
{
    "type": "HTTP",
    "httpMethod": "GET",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
    "connectionType": "INTERNET",
    "requestParameters": {
        "integration.request.path.id": "method.request.path.petId"
    },
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "rjkmth",
    "cacheKeyParameters": []
}
```

7. Call the `put-integration-response` command to create an `IntegrationResponse` of the `GET /pets` method integrated with an HTTP backend.

   ```
   aws apigateway put-integration-response --rest-api-id vaz7da96z6 \
          --resource-id 6sxz2j --http-method GET \
          --status-code 200 --selection-pattern ""  \
          --region us-west-2
   ```

   The following is the output of this command:

   ```
   {
       "statusCode": "200",
       "selectionPattern": ""
   }
   ```

   Similarly, call the following `put-integration-response` command to create an `IntegrationResponse` of the `GET /pets/{petId}` method:

   ```
   aws apigateway put-integration-response --rest-api-id vaz7da96z6 \
         --resource-id rjkmth --http-method GET
         --status-code 200 --selection-pattern ""
         --region us-west-2
   ```

   With the preceding steps, you finished setting up a simple API that allows your customers to query available pets on the PetStore website and to view an individual pet of a specified identifier. To make it callable by your customer, you must deploy the API.

8. Deploy the API to a `stage` stage, for example, by calling `create-deployment`:

   ```
   aws apigateway create-deployment --rest-api-id vaz7da96z6 \
          --region us-west-2 \
          --stage-name test \
          --stage-description 'Test stage' \
          --description 'First deployment'
   ```

   The following is the output of this command:

```
{
    "id": "ab1c1d",
    "description": "First deployment",
    "createdDate": "2022-12-15T08:44:13-08:00"
}
```

You can test this API by typing the `https://vaz7da96z6.execute-api.us-west-2.amazonaws.com/test/pets` URL in a browser, and substituting `vaz7da96z6` with the identifier of your API. The expected output should be as follows:

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

To test the `GET /pets/{petId}` method, type `https://vaz7da96z6.execute-api.us-west-2.amazonaws.com/test/pets/3` in the browser. You should receive the following response:

```
{
  "id": 3,
  "type": "fish",
  "price": 0.99
}
```

Set up an edge-optimized API using the AWS SDK for Node.js

As an illustration, we use AWS SDK for Node.js to describe how you can use an AWS SDK to create an API Gateway API. For more information using an AWS SDK, including how to set up the development environment, see AWS SDKs.

Setting up an API using the AWS SDK for Node.js involves calling the `createRestApi`, `createResource` or `getResources`, `putMethod`, `putMethodResponse`, `putIntegration`, and `putIntegrationResponse` functions.

The following procedures walk you through the essential steps to use these SDK commands to set up a simple PetStore API supporting the `GET /pets` and `GET /pets/{petId}` methods.

To set up a simple PetStore API using the AWS SDK for Node.js

1. Instantiate the SDK:

```
var AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
var apig = new AWS.APIGateway({apiVersion: '2015/07/09'});
```

2. Call the `createRestApi` function to set up the `RestApi` entity.

```
apig.createRestApi({
 name: "Simple PetStore (node.js SDK)",
 binaryMediaTypes: [
  '*'
 ],
 description: "Demo API created using the AWS SDK for node.js",
 version: "0.00.001"
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log('Create API failed:\n', err);
 }
});
```

The function returns an output similar to the following result:

```
{
  id: 'iuo308uaq7',
  name: 'PetStore (node.js SDK)',
  description: 'Demo API created using the AWS SDK for node.js',
  createdDate: 2017-09-05T19:32:35.000Z,
  version: '0.00.001',
  binaryMediaTypes: [ '*' ]
}
```

The resulting API's identifier is `iuo308uaq7`. You need to supply this to continue the setup of the API.

3. Call the `getResources` function to retrieve the root resource identifier of the `RestApi`.

```
apig.getResources({
 restApiId: 'iuo308uaq7'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log('Get the root resource failed:\n', err);
 }
})
```

This function returns an output similar to the following result:

```
{
    "items": [
        {
            "path": "/",
            "id": "s4fb0trnk0"
        }
    ]
}
```

The root resource identifier is `s4fb0trnk0`. This is the starting point for you to build the API resource tree, which you do next.

4. Call the `createResource` function to set up the `/pets` resource for the API, specifying the root resource identifier (`s4fb0trnk0`) on the `parentId` property.

```
apig.createResource({
 restApiId: 'iuo308uaq7',
 parentId: 's4fb0trnk0',
 pathPart: 'pets'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("The '/pets' resource setup failed:\n", err);
 }
})
```

The successful result is as follows:

```
{
    "path": "/pets",
    "pathPart": "pets",
    "id": "8sxa2j",
    "parentId": "s4fb0trnk0'"
}
```

To set up the `/pets/{petId}` resource, call the following `createResource` function, specifying the newly created `/pets` resource (`8sxa2j`) on the `parentId` property.

```
apig.createResource({
 restApiId: 'iuo308uaq7',
 parentId: '8sxa2j',
 pathPart: '{petId}'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("The '/pets/{petId}' resource setup failed:\n", err);
 }
})
```

The successful result returns the newly created resource `id` value:

```
{
    "path": "/pets/{petId}",
    "pathPart": "{petId}",
    "id": "au5df2",
    "parentId": "8sxa2j"
}
```

Throughout this procedure, you refer to the `/pets` resource by specifying its resource ID of `8sxa2j`, and the `/pets/{petId}` resource by specifying its resource ID of `au5df2`.

5. Call the `putMethod` function to add the `GET` HTTP method on the `/pets` resource (`8sxa2j`). This sets up the `GET /pets` `Method` with open access.

```
apig.putMethod({
 restApiId: 'iuo308uaq7',
 resourceId: '8sxa2j',
 httpMethod: 'GET',
 authorizationType: 'NONE'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("The 'GET /pets' method setup failed:\n", err);
 }
})
```

This function returns an output similar to the following result:

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE"
}
```

To add the `GET` HTTP method on the `/pets/{petId}` resource (`au5df2`), which sets up the API method of `GET /pets/{petId}` with open access, call the `putMethod` function as follows.

```
apig.putMethod({
 restApiId: 'iuo308uaq7',
 resourceId: 'au5df2',
 httpMethod: 'GET',
 authorizationType: 'NONE',
      requestParameters: {
        "method.request.path.petId" : true
      }
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("The 'GET /pets/{petId}' method setup failed:\n", err);
 }
})
```

This function returns an output similar to the following result:

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "requestParameters": {
       "method.request.path.petId": true
    }
}
```

You need to set the `requestParameters` property as shown in the preceding example to map and pass the client-supplied `petId` value to the backend.

6. Call the `putMethodResponse` function to set up a method response for the `GET /pets` method.

```
apig.putMethodResponse({
 restApiId: 'iuo308uaq7',
 resourceId: "8sxa2j",
 httpMethod: 'GET',
 statusCode: '200'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("Set up the 200 OK response for the 'GET /pets' method failed:\n",
 err);
 }

})
```

This function returns an output similar to the following result:

```
{
    "statusCode": "200"
}
```

To set the 200 OK response of the `GET /pets/{petId}` method, call the `putMethodResponse` function, specifying the `/pets/{petId}` resource identifier (`au5df2`) on the `resourceId` property.

```
apig.putMethodResponse({
 restApiId: 'iuo308uaq7',
 resourceId: "au5df2",
 httpMethod: 'GET',
 statusCode: '200'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("Set up the 200 OK response for the 'GET /pets/{petId}' method
 failed:\n", err);
 }

})
```

7. Call the `putIntegration` function to set up the `Integration` with a specified HTTP endpoint for the `GET /pets` method, supplying the `/pets` resource identifier (`8sxa2j`) on the `parentId` property.

```
apig.putIntegration({
 restApiId: 'iuo308uaq7',
 resourceId: '8sxa2j',
 httpMethod: 'GET',
 type: 'HTTP',
 integrationHttpMethod: 'GET',
 uri: 'http://perstore-demo-endpoint.execute-api.com/pets'
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("Set up the integration of the 'GET /' method of the API failed:\n",
 err);
 }
})
```

This function returns an output similar the following:

```
{
    "httpMethod": "GET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "HTTP",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
    "cacheNamespace": "8sxa2j"
}
```

To set up the integration of the `GET /pets/{petId}` method with the HTTP endpoint of `http://perstore-demo-endpoint.execute-api.com/pets/{id}` of the backend, call the following `putIntegration` function, supplying the API's `/pets/{petId}` resource identifier (`au5df2`) on the `parentId` property.

```
apig.putIntegration({
 restApiId: 'iuo308uaq7',
 resourceId: 'au5df2',
 httpMethod: 'GET',
 type: 'HTTP',
 integrationHttpMethod: 'GET',
 uri: 'http://perstore-demo-endpoint.execute-api.com/pets/{id}',
       requestParameters: {
          "integration.request.path.id": "method.request.path.petId"
       }
}, function(err, data){
 if (!err) {
  console.log(data);
 } else {
  console.log("The 'GET /pets/{petId}' method integration setup failed:\n", err);
 }

})
```

This function returns a successful output similar to the following:

```
{
    "httpMethod": "GET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "HTTP",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
    "cacheNamespace": "au5df2",
    "requestParameters": {
       "integration.request.path.id": "method.request.path.petId"
    }
}
```

8. Call the `putIntegrationResponse` function to set up the 200 OK integration response for the `GET /pets` method, specifying the API's `/pets` resource identifier (`8sxa2j`) on the `resourceId` property.

```
apig.putIntegrationResponse({
 restApiId: 'iuo308uaq7',
 resourceId: '8sxa2j',
 httpMethod: 'GET',
 statusCode: '200',
 selectionPattern: ''
}, function(err, data){
 if (!err) {
  console.log(data);
 } else
  console.log("The 'GET /pets' method integration response setup failed:\n", err);

})
```

This function will return an output similar to the following result:

```
{
    "selectionPattern": "",
    "statusCode": "200"
}
```

To set up the 200 OK integration response of the `GET /pets/{petId}` method, call the `putIntegrationResponse` function, specifying the API's `/pets/{petId}` resource identifier (`au5df2`) on the `resourceId` property.

```
apig.putIntegrationResponse({
 restApiId: 'iuo308uaq7',
 resourceId: 'au5df2',
 httpMethod: 'GET',
 statusCode: '200',
 selectionPattern: ''
}, function(err, data){
 if (!err) {
  console.log(data);
 } else
  console.log("The 'GET /pets/{petId}' method integration response setup failed:
\n", err);

})
```

9. As a good practice, test invoking the API before deploying it. To test invoking the `GET /pets` method, call the `testInvokeMethod`, specifying the `/pets`resource identifier (`8sxa2j`) on the `resourceId` property:

```
apig.testInvokeMethod({
 restApiId: 'iuo308uaq7',
 resourceId: '8sxa2j',
 httpMethod: "GET",
 pathWithQueryString: '/'
}, function(err, data){
 if (!err) {
  console.log(data)
 } else {
  console.log('Test-invoke-method on 'GET /pets' failed:\n', err);
 }
})
```

To test invoking the `GET /pets/{petId}` method, call the `testInvokeMethod`, specifying the `/pets/{petId}` resource identifier (`au5df2`) on the `resourceId` property:

```
apig.testInvokeMethod({
 restApiId: 'iuo308uaq7',
 resourceId: 'au5df2',
 httpMethod: "GET",
 pathWithQueryString: '/'
}, function(err, data){
 if (!err) {
  console.log(data)
 } else {
  console.log('Test-invoke-method on 'GET /pets/{petId}' failed:\n', err);
 }
})
```

10. Finally, you can deploy the API for your customers to call.

```
apig.createDeployment({
 restApiId: 'iuo308uaq7',
 stageName: 'test',
 stageDescription: 'test deployment',
 description: 'API deployment'
}, function(err, data){
 if (err) {
  console.log('Deploying API failed:\n', err);
 } else {
  console.log("Deploying API succeeded\n", data);
 }
})
```

Set up an edge-optimized API by importing OpenAPI definitions

You can set up an API in API Gateway by specifying OpenAPI definitions of appropriate API Gateway API entities and importing the OpenAPI definitions into API Gateway.

The following OpenAPI definitions describe the simple API, exposing only the `GET /` method integrated with an HTTP endpoint of the PetStore website in the backend, and returning a `200 OK` response.

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "title": "Simple PetStore (OpenAPI)"
  },
  "schemes": [
    "https"
  ],
  "paths": {
    "/pets": {
        "get": {
          "responses": {
            "200": {
              "description": "200 response"
            }
          },
          "x-amazon-apigateway-integration": {
            "responses": {
              "default": {
                "statusCode": "200"
              }
            },
            "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
            "passthroughBehavior": "when_no_match",
            "httpMethod": "GET",
            "type": "http"
          }
        }
      },
      "/pets/{petId}": {
        "get": {
          "parameters": [
            {
              "name": "petId",
              "in": "path",
              "required": true,
              "type": "string"
            }
          ],
          "responses": {
            "200": {
              "description": "200 response"
            }
          },
          "x-amazon-apigateway-integration": {
            "responses": {
              "default": {
                "statusCode": "200"
              }
            },
            "requestParameters": {
              "integration.request.path.id": "method.request.path.petId"
            },
            "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
            "passthroughBehavior": "when_no_match",
            "httpMethod": "GET",
            "type": "http"
          }
        }
      }
    }
  }
```

The following procedure describes how to import these OpenAPI definitions into API Gateway using the API Gateway console.

To import the simple OpenAPI definitions using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**, and then for **REST API**, choose **Import**.
3. If you saved the preceding OpenAPI definitions in a file, choose **Choose file**. You can also copy the OpenAPI definitions and paste them into the import text editor.
4. For **API endpoint type**, select **Edge-optimized**.
5. Choose **Create API** to import the OpenAPI definitions.

To import the OpenAPI definitions using the AWS CLI, save the OpenAPI definitions into a file and then run the following command, assuming that you use the `us-west-2` region and the absolute OpenAPI file path is `file:///path/to/API_OpenAPI_template.json`:

```
aws apigateway import-rest-api --body 'file:///path/to/API_OpenAPI_template.json' --
region us-west-2
```

Set up a Regional API in API Gateway

When API requests predominantly originate from an EC2 instance or services within the same region as the API is deployed, a Regional API endpoint will typically lower the latency of connections and is recommended for such scenarios.

> **Note**
>
> In cases where API clients are geographically dispersed, it may still make sense to use a Regional API endpoint, together with your own Amazon CloudFront distribution to

ensure that API Gateway does not associate the API with service-controlled CloudFront distributions. For more information about this use case, see How do I set up API Gateway with my own CloudFront distribution?.

To create a Regional API, you follow the steps in creating an edge-optimized API, but must explicitly set `REGIONAL` type as the only option of the API's endpointConfiguration.

In the following, we show how to create a Regional API using the API Gateway console, AWS CLI, and the AWS SDK for Javascript for Node.js.

Topics

Create a Regional API using the API Gateway console

To create a Regional API using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
   - To create your first API, for **REST API**, choose **Build**.
   - If you've created an API before, choose **Create API**, and then choose **Build** for **REST API**.
3. For **Name**, enter a name.
4. (Optional) For **Description**, enter a description.
5. Keep **API endpoint type** set to **Regional**.
6. Choose **Create API**.

Create a Regional API using the AWS CLI

To create a Regional API using the AWS CLI, call the `create-rest-api` command:

```
aws apigateway create-rest-api \
        --name 'Simple PetStore (AWS CLI, Regional)' \
        --description 'Simple regional PetStore API' \
        --region us-west-2 \
        --endpoint-configuration '{ "types": ["REGIONAL"] }'
```

A successful response returns a payload similar to the following:

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "Simple regional PetStore API",
    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (AWS CLI, Regional)"
}
```

From here on, you can follow the same instructions given in the section called "Set up an edge-optimized API using AWS CLI commands" to set up methods and integrations for this API.

Create a Regional API using the AWS SDK for JavaScript

To create a Regional API, using the AWS SDK for JavaScript:

```
apig.createRestApi({
 name: "Simple PetStore (node.js SDK, regional)",
 endpointConfiguration: {
  types: ['REGIONAL']
 },
 description: "Demo regional API created using the AWS SDK for node.js",
 version: "0.00.001"
}, function(err, data){
 if (!err) {
  console.log('Create API succeeded:\n', data);
  restApiId = data.id;
 } else {
  console.log('Create API failed:\n', err);
 }
});
```

A successful response returns a payload similar to the following:

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "Demo regional API created using the AWS SDK for node.js",
    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (node.js SDK, regional)"
}
```

After completing the preceding steps, you can follow the instructions in the section called "Set up an edge-optimized API using the AWS SDK for Node.js" to set up methods and integrations for this API.

Create a Regional API using an OpenAPI definition

To import an API from an OpenAPI definition file using the AWS CLI, use the `import-rest-api` command:

```
aws apigateway import-rest-api \
    --parameters endpointConfigurationTypes=REGIONAL \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

Test a Regional API

Once deployed, the Regional API's default URL host name is of the following format:

```
{restapi-id}.execute-api.{region}.amazonaws.com
```

The base URL to invoke the API is like the following:

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stage}
```

Assuming you set up the `GET /pets` and `GET /pets/{petId}` methods in this example, you can test the API by typing the following URLs in a browser:

```
https://0qzs2sy7bh.execute-api.us-west-2.amazonaws.com/test/pets
```

and

```
https://0qzs2sy7bh.execute-api.us-west-2.amazonaws.com/test/pets/1
```

Alternatively, you can use cURL commands:

```
curl -X GET https://0qzs2sy7bh.execute-api.us-west-2.amazonaws.com/test/pets
```

and

```
curl -X GET https://0qzs2sy7bh.execute-api.us-west-2.amazonaws.com/test/pets/2
```

Set up REST API methods in API Gateway

In API Gateway, an API method embodies a method request and a method response. You set up an API method to define what a client should or must do to submit a request to access the service at the backend and to define the responses that the client receives in return. For input, you can choose method request parameters, or an applicable payload, for the client to provide the required or optional data at run time. For output, you determine the method response status code, headers, and applicable body as targets to map the backend response data into, before they are returned to the client. To help the client developer understand the behaviors and the input and output formats of your API, you can document your API and provide proper error messages for invalid requests.

An API method request is an HTTP request. To set up the method request, you configure an HTTP method (or verb), the path to an API resource, headers, applicable query string parameters. You also configure a payload when the HTTP method is `POST`, `PUT`, or `PATCH`. For example, to retrieve a pet using the PetStore sample API, you define the API method request of `GET /pets/{petId}`, where `{petId}` is a path parameter that can take a number at run time.

```
GET /pets/1
Host: apigateway.us-east-1.amazonaws.com
...
```

If the client specifies an incorrect path, for example, `/pet/1` or `/pets/one` instead of `/pets/1`, an exception is thrown.

An API method response is an HTTP response with a given status code. For a non-proxy integration, you must set up method responses to specify the required or optional targets of mappings. These transform integration response headers or body to associated method response headers or body. The mapping can be as simple as an identity transform that passes the headers or body through the integration as-is. For example, the following `200` method response shows an example of passthrough of a successful integration response as-is.

```
200 OK
Content-Type: application/json
...

{
    "id": "1",
    "type": "dog",
    "price": "$249.99"
}
```

In principle, you can define a method response corresponding to a specific response from the backend. Typically, this involves any 2XX, 4XX, and 5XX responses. However, this may not be practical, because often you may not know in advance all the responses that a backend may return. In practice, you can designate one method response as the default to handle the unknown or unmapped responses from the backend. It is good practice to designate the 500 response as the default. In any case, you must set up at least one method response for non-proxy integrations. Otherwise, API Gateway returns a 500 error response to the client even when the request succeeds at the backend.

To support a strongly typed SDK, such as a Java SDK, for your API, you should define the data model for input for the method request, and define the data model for output of the method response.

Prerequisites

Before setting up an API method, verify the following:

- You must have the method available in API Gateway. Follow the instructions in Tutorial: Build a REST API with HTTP non-proxy integration.
- If you want the method to communicate with a Lambda function, you must have already created the Lambda invocation role and Lambda execution role in IAM. You must also have created the Lambda function with which your method will communicate in AWS Lambda. To create the roles and function, use the instructions in Create a Lambda function for Lambda non-proxy integration of the Build an API Gateway REST API with Lambda integration.

- If you want the method to communicate with an HTTP or HTTP proxy integration, you must have already created, and have access to, the HTTP endpoint URL with which your method will communicate.
- Verify that your certificates for HTTP and HTTP proxy endpoints are supported by API Gateway. For details see API Gateway-supported certificate authorities for HTTP and HTTP proxy integrations.

Note

When you create a method using the REST API console, you configure both the integration request and the method request. For more information, see the section called " Set up integration request using the console".

Topics

- Set up a method request in API Gateway
- Set up method responses in API Gateway
- Set up a method using the API Gateway console

Set up a method request in API Gateway

Setting up a method request involves performing the following tasks, after creating a RestApi resource:

1. Creating a new API or choosing an existing API Resource entity.
2. Creating an API Method resource that is a specific HTTP verb on the new or chosen API `Resource`. This task can be further divided into the following sub tasks:
    - Adding an HTTP method to the method request
    - Configuring request parameters
    - Defining a model for the request body
    - Enacting an authorization scheme
    - Enabling request validation

You can perform these tasks using the following methods:

- API Gateway console
- AWS CLI commands (create-resource and put-method)
- AWS SDK functions (for example, in Node.js, createResource and putMethod)
- API Gateway REST API (resource:create and method:put).

For examples of using these tools, see Initialize REST API setup in API Gateway.

Topics

Set up API resources

In an API Gateway API, you expose addressable resources as a tree of API Resources entities, with the root resource (/) at the top of the hierarchy. The root resource is relative to the API's base URL, which consists of the API endpoint and a stage name. In the API Gateway console, this base URI is referred to as the **Invoke URI** and is displayed in the API's stage editor after the API is deployed.

The API endpoint can be a default host name or a custom domain name. The default host name is of the following format:

```
{api-id}.execute-api.{region}.amazonaws.com
```

In this format, the *{api-id}* represents the API identifier that is generated by API Gateway. The *{region}* variable represents the AWS Region (for example, us-east-1) that you chose when creating the API. A custom domain name is any user-friendly name under a valid internet domain. For example, if you have registered an internet domain of example.com, any of *.example.com is a valid custom domain name. For more information, see create a custom domain name.

For the PetStore sample API, the root resource (/) exposes the pet store. The /pets resource represents the collection of pets available in the pet store. The /pets/{petId} exposes an individual pet of a given identifier (petId). The path parameter of {petId} is part of the request parameters.

To set up an API resource, you choose an existing resource as its parent and then create the child resource under this parent resource. You start with the root resource as a parent, add a resource to this parent, add another resource to this child resource as the new parent, and so on, to its parent identifier. Then you add the named resource to the parent.

With AWS CLI, you can call the get-resources command to find out which resources of an API are available:

```
aws apigateway get-resources --rest-api-id <apiId> \
                             --region <region>
```

The result is a list of the currently available resources of the API. For our PetStore sample API, this list looks like the following:

```
{
    "items": [
        {
            "path": "/pets",
            "resourceMethods": {
                "GET": {}
            },
            "id": "6sxz2j",
            "pathPart": "pets",
            "parentId": "svzr2028x8"
        },
        {
            "path": "/pets/{petId}",
            "resourceMethods": {
                "GET": {}
            },
            "id": "rjkmth",
            "pathPart": "{petId}",
            "parentId": "6sxz2j"
        },
        {
            "path": "/",
            "id": "svzr2028x8"
        }
    ]
}
```

Each item lists the identifiers of the resource (`id`) and, except for the root resource, its immediate parent (`parentId`), as well as the resource name (`pathPart`). The root resource is special in that it does not have any parent. After choosing a resource as the parent, call the following command to add a child resource.

```
aws apigateway create-resource --rest-api-id <apiId> \
                               --region <region> \
                               --parent-id <parentId> \
                               --path-part <resourceName>
```

For example, to add pet food for sale on the PetStore website, add a `food` resource to the root (/) by setting `path-part` to `food` and `parent-id` to `svzr2028x8`. The result looks like the following:

```
{
    "path": "/food",
    "pathPart": "food",
    "id": "xdsvhp",
    "parentId": "svzr2028x8"
}
```

Use a proxy resource to streamline API setup

As business grows, the PetStore owner may decide to add food, toys, and other pet-related items for sale. To support this, you can add `/food`, `/toys`, and other resources under the root resource. Under each sale category, you may also want to add more resources, such as `/food/{type}/{item}`, `/toys/{type}/{item}`, etc. This can get tedious. If you decide to add a middle layer `{subtype}` to the resource paths to change the path hierarchy into `/food/{type}/{subtype}/{item}`, `/toys/{type}/{subtype}/{item}`, etc., the changes will break the existing API set up. To avoid this, you can use an API Gateway proxy resource to expose a set of API resources all at once.

API Gateway defines a proxy resource as a placeholder for a resource to be specified when the request is submitted. A proxy resource is expressed by a special path parameter of `{proxy+}`, often referred to as a greedy path parameter. The + sign indicates whichever child resources are appended to it. The `/parent/{proxy+}` placeholder stands for any resource matching the path pattern of /parent/*. The greedy path parameter name, proxy, can be replaced by another string in the same way you treat a regular path parameter name.

Using the AWS CLI, you call the following command to set up a proxy resource under the root (/{proxy+}):

```
aws apigateway create-resource --rest-api-id <apiId> \
                               --region <region> \
                               --parent-id <rootResourceId> \
                               --path-part {proxy+}
```

The result is similar to the following:

```
{
    "path": "/{proxy+}",
    "pathPart": "{proxy+}",
    "id": "234jdr",
    "parentId": "svzr2028x8"
}
```

For the PetStore API example, you can use /{proxy+} to represent both the /pets and /pets/{petId}. This proxy resource can also reference any other (existing or to-be-added) resources, such as /food/{type}/{item}, /toys/{type}/{item}, etc., or /food/{type}/{subtype}/{item}, /toys/{type}/{subtype}/{item}, etc. The backend developer determines the resource hierarchy and the client developer is responsible for understanding it. API Gateway simply passes whatever the client submitted to the backend.

An API can have more than one proxy resource. For example, the following proxy resources are allowed within an API.

```
/{proxy+}
/parent/{proxy+}
/parent/{child}/{proxy+}
```

When a proxy resource has non-proxy siblings, the sibling resources are excluded from the representation of the proxy resource. For the preceding examples, /{proxy+} refers to any resources under the root resource except for the /parent[/*] resources. In other words, a method request against a specific resource takes precedence over a method request against a generic resource at the same level of the resource hierarchy.

A proxy resource cannot have any child resource. Any API resource after {proxy+} is redundant and ambiguous. The following proxy resources are not allowed within an API.

```
/{proxy+}/child
/parent/{proxy+}/{child}
/parent/{child}/{proxy+}/{grandchild+}
```

Set up an HTTP method

An API method request is encapsulated by the API Gateway Method resource. To set up the method request, you must first instantiate the Method resource, setting at least an HTTP method and an authorization type on the method.

Closely associated with the proxy resource, API Gateway supports an HTTP method of ANY. This ANY method represents any HTTP method that is to be supplied at run time. It allows you to use a single API method setup for all of the supported HTTP methods of DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT.

You can set up the ANY method on a non-proxy resource as well. Combining the ANY method with a proxy resource, you get a single API method setup for all of the supported HTTP methods against any resources of an API. Furthermore, the backend can evolve without breaking the existing API setup.

Before setting up an API method, consider who can call the method. Set the authorization type according to your plan. For open access, set it to NONE. To use IAM permissions, set the authorization type to AWS_IAM. To use a Lambda authorizer function, set this property to CUSTOM. To use an Amazon Cognito user pool, set the authorization type to COGNITO_USER_POOLS.

The following AWS CLI command shows how to create a method request of the ANY verb against a specified resource (6sxz2j), using the IAM permissions to control its access.

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method ANY \
       --authorization-type AWS_IAM \
       --region us-west-2
```

To create an API method request with a different authorization type, see the section called “Set up method request authorization”.

Set up method request parameters

Method request parameters are a way for a client to provide input data or execution context necessary to complete the method request. A method parameter can be a path parameter, a header, or a query string parameter. As part of method request setup, you must declare required request parameters to make them available for the client. For non-proxy integration, you can translate these request parameters to a form that is compatible with the backend requirement.

For example, for the `GET /pets/{petId}` method request, the `{petId}` path variable is a required request parameter. You can declare this path parameter when calling the `put-method` command of the AWS CLI. This is illustrated as follows:

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id rjkmth \
       --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-parameters method.request.path.petId=true
```

If a parameter is not required, you can set it to `false` in `request-parameters`. For example, if the `GET /pets` method uses an optional query string parameter of `type`, and an optional header parameter of `breed`, you can declare them using the following CLI command, assuming that the `/pets` resource `id` is `6sxz2j`:

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-parameters
 method.request.querystring.type=false,method.request.header.breed=false
```

Instead of this abbreviated form, you can use a JSON string to set the `request-parameters` value:

```
'{"method.request.querystring.type":false,"method.request.header.breed":false}'
```

With this setup, the client can query pets by type:

```
GET /pets?type=dog
```

And the client can query dogs of the poodle breed as follows:

```
GET /pets?type=dog
breed:poodle
```

For information on how to map method request parameters to integration request parameters, see the section called “Integrations”.

Set up method request model

For an API method that can take input data in a payload, you can use a model. A model is expressed in a JSON schema draft 4 and describes the data structure of the request body. With a model, a client can determine how to construct a method request payload as input. More importantly, API Gateway uses the model to validate a request, generate an SDK, and initialize a mapping template for setting up the integration in the API Gateway console. For information about how to create a model, see Understanding data models.

Depending on the content types, a method payload can have different formats. A model is indexed against the media type of the applied payload. API Gateway uses the `Content-Type` request header to determine the content type. To set up method request models, add key-value pairs of the `"`*`<media-type>`*`":"`*`<model-name>`*`"` format to the `requestModels` map when calling the AWS CLI `put-method` command.

To use the same model regardless of the content type, specify `$default` as the key.

For example, to set a model on the JSON payload of the `POST /pets` method request of the PetStore example API, you can call the following AWS CLI command:

```
aws apigateway put-method \
       --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method POST \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-models '{"application/json":"petModel"}'
```

Here, `petModel` is the `name` property value of a `Model` resource describing a pet. The actual schema definition is expressed as a JSON string value of the `schema` property of the `Model` resource.

In a Java, or other strongly typed SDK, of the API, the input data is cast as the `petModel` class derived from the schema definition. With the request model, the input data in the generated SDK is cast into the `Empty` class, which is derived from the default `Empty` model. In this case, the client cannot instantiate the correct data class to provide the required input.

Set up method request authorization

To control who can call the API method, you can configure the authorization type on the method. You can use this type to enact one of the supported authorizers, including IAM roles and policies (`AWS_IAM`), an Amazon Cognito user pool (`COGNITO_USER_POOLS`), or a Lambda authorizer (`CUSTOM`).

To use IAM permissions to authorize access to the API method, set the `authorization-type` input property to **`AWS_IAM`**. When you set this option, API Gateway verifies the caller's signature on the request based on the caller's credentials. If the verified user has permission to call the method, it accepts the request. Otherwise, it rejects the request and the caller receives an unauthorized error response. The call to the method doesn't succeed unless the caller has permission to invoke the API method. The following IAM policy grants permission to the caller to call any API methods created within the same AWS account:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke"
            ],
            "Resource": "arn:aws:execute-api:*:*:*"
        }
    ]
}
```

For more information, see the section called “Use IAM permissions”.

Currently, you can only grant this policy to the users, groups, and roles within the API owner's AWS account. Users from a different AWS account can call the API methods only if allowed to assume a role within the API owner's AWS account with the necessary permissions to call the `execute-api:Invoke` action. For information on cross-account permissions, see Using IAM Roles.

You can use AWS CLI, an AWS SDK, or a REST API client, such as Postman, which implements Signature Version 4 Signing.

To use a Lambda authorizer to authorize access to the API method, set the `authorization-type` input property to `CUSTOM` and set the `authorizer-id` input property to the `id` property value of a Lambda authorizer that already exists. The referenced Lambda authorizer can be of the `TOKEN` or `REQUEST` type. For information about creating a Lambda authorizer, see the section called "Use Lambda authorizers".

To use an Amazon Cognito user pool to authorize access to the API method, set the `authorization-type` input property to `COGNITO_USER_POOLS` and set the `authorizer-id` input property to the `id` property value of the `COGNITO_USER_POOLS` authorizer that was already created. For information about creating an Amazon Cognito user pool authorizer, see the section called "Use Amazon Cognito user pool as authorizer for a REST API".

Set up method request validation

You can enable request validation when setting up an API method request. You need to first create a request validator:

```
aws apigateway create-request-validator \
    --rest-api-id 7zw9uyk9kl \
    --name bodyOnlyValidator \
    --validate-request-body  \
    --no-validate-request-parameters
```

This CLI command creates a body-only request validator. Example output is as follows:

```
{
    "validateRequestParameters": false,
    "validateRequestBody": true,
    "id": "jgpyy6",
    "name": "bodyOnlyValidator"
}
```

With this request validator, you can enable request validation as part of the method request setup:

```
aws apigateway put-method \
    --rest-api-id 7zw9uyk9kl
    --region us-west-2
    --resource-id xdsvhp
    --http-method PUT
    --authorization-type "NONE"
    --request-parameters '{"method.request.querystring.type": false,
 "method.request.querystring.page":false}'
    --request-models '{"application/json":"petModel"}'
    --request-validator-id jgpyy6
```

To be included in request validation, a request parameter must be declared as required. If the query string parameter for the page is used in request validation, the `request-parameters` map of the preceding example must be specified as `'{"method.request.querystring.type": false, "method.request.querystring.page":true}'`.

Set up method responses in API Gateway

An API method response encapsulates the output of an API method request that the client will receive. The output data includes an HTTP status code, some headers, and possibly a body.

With non-proxy integrations, the specified response parameters and body can be mapped from the associated integration response data or can be assigned certain static values according to mappings. These mappings are specified in the integration response. The mapping can be an identical transformation that passes the integration response through as-is.

With a proxy integration, API Gateway passes the backend response through to the method response automatically. There is no need for you to set up the API method response. However, with the Lambda proxy integration, the Lambda function must return a result of this output format for API Gateway to successfully map the integration response to a method response.

Programmatically, the method response setup amounts to creating a MethodResponse resource of API Gateway and setting the properties of statusCode, responseParameters, and responseModels.

When setting status codes for an API method, you should choose one as the default to handle any integration response of an unanticipated status code. It is reasonable to set `500` as the default because this amounts to casting otherwise unmapped responses as a server-side error. For instructional reasons, the API Gateway console sets the `200` response as the default. But you can reset it to the `500` response.

To set up a method response, you must have created the method request.

Set up method response status code

The status code of a method response defines a type of response. For example, responses of 200, 400, and 500 indicate successful, client-side error and server-side error responses, respectively.

To set up a method response status code, set the `statusCode` property to an HTTP status code. The following AWS CLI command creates a method response of `200`.

```
aws apigateway put-method-response \
       --region us-west-2  \
       --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method GET \
       --status-code 200
```

Set up method response parameters

Method response parameters define which headers the client receives in response to the associated method request. Response parameters also specify a target to which API Gateway maps an integration response parameter, according to mappings prescribed in the API method's integration response.

To set up the method response parameters, add to the `responseParameters` map of `MethodResponse` key-value pairs of the `"{parameter-name}":"{boolean}"` format. The following CLI command shows an example of setting the `my-header` header.

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id vaz7da96z6 \
        --resource-id 6sxz2j \
        --http-method GET \
        --status-code 200  \
        --response-parameters method.response.header.my-header=false
```

Set up method response models

A method response model defines a format of the method response body. Before setting up the response model, you must first create the model in API Gateway. To do so, you can call the `create-model` command. The following example shows how to create a `PetStorePet` model to describe the body of the response to the `GET /pets/{petId}` method request.

```
aws apigateway create-model \
    --region us-west-2 \
    --rest-api-id vaz7da96z6 \
    --content-type application/json \
    --name PetStorePet \
    --schema '{ \
                  "$schema": "http://json-schema.org/draft-04/schema#", \
                  "title": "PetStorePet", \
                  "type": "object", \
                  "properties": { \
                    "id": { "type": "number" }, \
                    "type": { "type": "string" }, \
                    "price": { "type": "number" } \
                  } \
              }'
```

The result is created as an API Gateway `Model` resource.

To set up the method response models to define the payload format, add the "application/json":"PetStorePet" key-value pair to the `requestModels` map of `MethodResponse` resource. The following AWS CLI command of `put-method-response` shows how this is done:

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id vaz7da96z6 \
        --resource-id 6sxz2j \
        --http-method GET \
        --status-code 200  \
        --response-parameters method.response.header.my-header=false \
        --response-models '{"application/json":"PetStorePet"}'
```

Setting up a method response model is necessary when you generate a strongly typed SDK for the API. It ensures that the output is cast into an appropriate class in Java or Objective-C. In other cases, setting a model is optional.

Set up a method using the API Gateway console

When you create a method using the REST API console, you configure both the integration request and the method request. By default, API Gateway creates the `200` method response for your method.

The following instructions show how to edit the method request settings and how to create additional method responses for your method.

Topics

- Edit an API Gateway method request in the API Gateway console

- Set up an API Gateway method response using the API Gateway console

Edit an API Gateway method request in the API Gateway console

These instructions assume you have already created your method request. For more information on how to create a method, see the section called “ Set up integration request using the console”.

1. In the **Resources** pane, choose your method, and then choose the **Method request** tab.
2. In the **Method request settings** section, choose **Edit**.
3. For **Authorization**, select an available authorizer.

    a. To enable open access to the method for any user, select **None**. This step can be skipped if the default setting has not been changed.

    b. To use IAM permissions to control the client access to the method, select `AWS_IAM`. With this choice, only users of the IAM roles with the correct IAM policy attached are allowed to call this method.

    To create the IAM role, specify an access policy with a format like the following:

    ```
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Action": [
            "execute-api:Invoke"
          ],
          "Resource": [
            "resource-statement"
          ]
        }
      ]
    }
    ```

    In this access policy, *`resource-statement`* is the ARN of your method. You can find the ARN of your method by selecting the method on the **Resources** page. For more information about setting the IAM permissions, see Control access to an API with IAM permissions.

    To create the IAM role, you can adapt the instructions in the following tutorial, ???.

   c. To use a Lambda authorizer, select a token or a request authorizer. Create the Lambda authorizer to have this choice displayed in the dropdown menu. For information on how to create a Lambda authorizer, see Use API Gateway Lambda authorizers.
   d. To use an Amazon Cognito user pool, choose an available user pool under **Cognito user pool authorizers**. Create a user pool in Amazon Cognito and an Amazon Cognito user pool authorizer in API Gateway to have this choice displayed in the dropdown menu. For information on how to create an Amazon Cognito user pool authorizer, see Control access to a REST API using Amazon Cognito user pools as authorizer.
4. To specify request validation, select a value from the **Request Validator** dropdown menu. To turn off request validation, select **None**. For more information about each option, see Use request validation in API Gateway.
5. Select **API key required** to require an API key. When enabled, API keys are used in usage plans to throttle client traffic.
6. (Optional) To assign an operation name in a Java SDK of this API, generated by API Gateway, for **Operation name**, enter a name. For example, for the method request of `GET /pets/{petId}`, the corresponding Java SDK operation name is, by default ,`GetPetsPetId`. This name is constructed from the method's HTTP verb (`GET`) and the resource path variable names (`Pets` and `PetId`). If you set the operation name as `getPetById`, the SDK operation name becomes `GetPetById`.
7. To add a query string parameter to the method, do the following:

   a. Choose **URL Query string parameters**, and then choose **Add query string**.
   b. For **Name**, enter the name of the query string parameter.
   c. Select **Required** if the newly created query string parameter is to be used for request validation. For more information about the request validation, see Use request validation in API Gateway.
   d. Select **Caching** if the newly created query string parameter is to be used as part of a caching key. For more information about caching, see Use method or integration parameters as cache keys to index cached responses.

   To remove the query string parameter, choose **Remove**.
8. To add a header parameter to the method, do the following:

   a. Choose **HTTP request headers**, and then choose **Add header**.
   b. For **Name**, enter the name of the header.

   c. Select **Required** if the newly created header is to be used for request validation. For more information about the request validation, see Use request validation in API Gateway.

   d. Select **Caching** if the newly created header is to be used as part of a caching key. For more information about caching, see Use method or integration parameters as cache keys to index cached responses.

   To remove the header, choose **Remove**.

9. To declare the payload format of a method request with the `POST`, `PUT`, or `PATCH` HTTP verb, choose **Request body**, and do the following:

   a. Choose **Add model**.

   b. For **Content-type**, enter a MIME-type (for example, `application/json`).

   c. For **Model**, select a model from the dropdown menu. The currently available models for the API include the default `Empty` and `Error` models as well as any models you have created and added to the Models collection of the API. For more information about creating a model, see Understanding data models.

   > **Note**
   >
   > The model is useful to inform the client of the expected data format of a payload. It is helpful to generate a skeletal mapping template. It is important to generate a strongly typed SDK of the API in such languages as Java, C#, Objective-C, and Swift. It is only required if request validation is enabled against the payload.

10. Choose **Save**.

Set up an API Gateway method response using the API Gateway console

An API method can have one or more responses. Each response is indexed by its HTTP status code. By default, the API Gateway console adds `200` response to the method responses. You can modify it, for example, to have the method return `201` instead. You can add other responses, for example, `409` for access denial and `500` for uninitialized stage variables used.

To use the API Gateway console to modify, delete, or add a response to an API method, follow these instructions.

1. In the **Resources** pane, choose your method, and then choose the **Method response** tab. You might need to choose the right arrow button to show the tab.
2. In the **Method response settings** section, choose **Create response**.
3. For **HTTP status code**, enter an HTTP status code such as `200`, `400`, or `500`.

   When a backend-returned response does not have a corresponding method response defined, API Gateway fails to return the response to the client. Instead, it returns a `500 Internal server error` error response.
4. Choose **Add header**.
5. For **Header name**, enter a name.

   To return a header from the backend to the client, add the header in the method response.
6. Choose **Add model** to define a format of the method response body.

   Enter the media type of the response payload for **Content type** and choose a model from the **Models** dropdown menu.
7. Choose **Save**.

To modify an existing response, navigate to your method response, and then choose **Edit**. To change the **HTTP status code**, choose **Delete** and create a new method response.

For every response returned from the backend, you must have a compatible response configured as the method response. However, the configuring method response headers and payload model are optional unless you map the result from the backend to the method response before returning to the client. Also, a method response payload model is important if you are generating a strongly typed SDK for your API.

Controlling and managing access to a REST API in API Gateway

API Gateway supports multiple mechanisms for controlling and managing access to your API.

You can use the following mechanisms for authentication and authorization:

- **Resource policies** let you create resource-based policies to allow or deny access to your APIs and methods from specified source IP addresses or VPC endpoints. For more information, see the section called "Use API Gateway resource policies".
- **Standard AWS IAM roles and policies** offer flexible and robust access controls that can be applied to an entire API or individual methods. IAM roles and policies can be used for controlling

who can create and manage your APIs, as well as who can invoke them. For more information, see the section called "Use IAM permissions".

- **IAM tags** can be used together with IAM policies to control access. For more information, see the section called "Attribute-based access control".
- **Endpoint policies for interface VPC endpoints** allow you to attach IAM resource policies to interface VPC endpoints to improve the security of your private APIs. For more information, see the section called "Use VPC endpoint policies for private APIs".
- **Lambda authorizers** are Lambda functions that control access to REST API methods using bearer token authentication—as well as information described by headers, paths, query strings, stage variables, or context variables request parameters. Lambda authorizers are used to control who can invoke REST API methods. For more information, see the section called "Use Lambda authorizers".
- **Amazon Cognito user pools** let you create customizable authentication and authorization solutions for your REST APIs. Amazon Cognito user pools are used to control who can invoke REST API methods. For more information, see the section called "Use Amazon Cognito user pool as authorizer for a REST API".

You can use the following mechanisms for performing other tasks related to access control:

- **Cross-origin resource sharing (CORS)** lets you control how your REST API responds to cross-domain resource requests. For more information, see the section called "CORS".
- **Client-side SSL certificates** can be used to verify that HTTP requests to your backend system are from API Gateway. For more information, see the section called "Client certificates".
- **AWS WAF** can be used to protect your API Gateway API from common web exploits. For more information, see the section called "AWS WAF".

You can use the following mechanisms for tracking and limiting the access that you have granted to authorized clients:

- **Usage plans** let you provide **API keys** to your customers—and then track and limit usage of your API stages and methods for each API key. For more information, see the section called "Usage plans".

Controlling access to an API with API Gateway resource policies

Amazon API Gateway *resource policies* are JSON policy documents that you attach to an API to control whether a specified principal (typically an IAM role or group) can invoke the API. You can use API Gateway resource policies to allow your API to be securely invoked by:

- Users from a specified AWS account.
- Specified source IP address ranges or CIDR blocks.
- Specified virtual private clouds (VPCs) or VPC endpoints (in any account).

You can use resource policies for all API endpoint types in API Gateway: private, edge-optimized, and Regional.

For private APIs, you can use resource policies together with VPC endpoint policies to control which principals have access to which resources and actions. For more information, see the section called “Use VPC endpoint policies for private APIs”.

You can attach a resource policy to an API by using the AWS Management Console, AWS CLI, or AWS SDKs.

API Gateway resource policies are different from IAM identity-based policies. IAM identity-based policies are attached to IAM users, groups, or roles and define what actions those identities are capable of doing on which resources. API Gateway resource policies are attached to resources. For a more detailed discussion of the differences between identity-based policies and resource policies, see Identity-Based Policies and Resource-Based Policies.

You can use API Gateway resource policies together with IAM policies.

Topics

Access policy language overview for Amazon API Gateway

This page describes the basic elements used in Amazon API Gateway resource policies.

Resource policies are specified using the same syntax as IAM policies. For complete policy language information, see [Overview of IAM Policies] and [AWS Identity and Access Management Policy Reference] in the *IAM User Guide*.

For information about how an AWS service decides whether a given request should be allowed or denied, see [Determining Whether a Request is Allowed or Denied].

Common elements in an access policy

In its most basic sense, a resource policy contains the following elements:

- **Resources** – APIs are the Amazon API Gateway resources for which you can allow or deny permissions. In a policy, you use the Amazon Resource Name (ARN) to identify the resource. You can also use abbreviated syntax, which API Gateway automatically expands to the full ARN when you save a resource policy. To learn more, see [API Gateway resource policy examples].

  For the format of the full `Resource` element, see [Resource format of permissions for executing API in API Gateway].
- **Actions** – For each resource, Amazon API Gateway supports a set of operations. You identify resource operations that you will allow (or deny) by using action keywords.

  For example, the `execute-api:Invoke` permission will allow the user permission to invoke an API upon a client request.

  For the format of the `Action` element, see [Action format of permissions for executing API in API Gateway].
- **Effect** – What the effect is when the user requests the specific action—this can be either `Allow` or `Deny`. You can also explicitly deny access to a resource, which you might do in order to make sure that a user cannot access it, even if a different policy grants access.

  > **Note**
  >
  > "Implicit deny" is the same thing as "deny by default".
  > An "implicit deny" is different from an "explicit deny". For more information, see [The Difference Between Denying by Default and Explicit Deny].
- **Principal** – The account or user allowed access to the actions and resources in the statement. In a resource policy, the principal is the user or account who receives this permission.

The following example resource policy shows the previous common policy elements. The policy grants access to the API under the specified *account-id* in the specified *region* to any user whose source IP address is in the address block *123.4.5.6/24*. The policy denies all access to the API if the user's source IP is not within the range.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:*"
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:*",
            "Condition": {
                "NotIpAddress": {
                    "aws:SourceIp": "123.4.5.6/24"
                }
            }
        }
    ]
}
```

How API Gateway resource policies affect authorization workflow

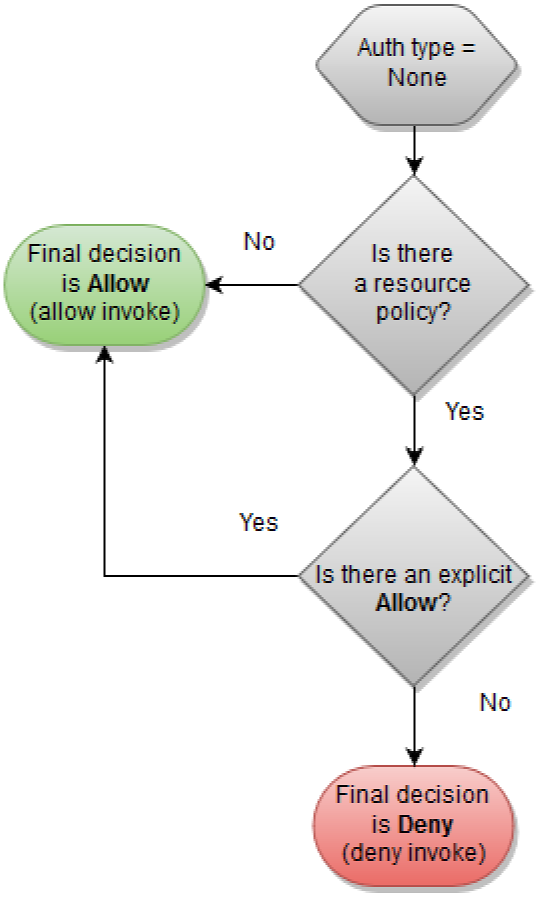
When API Gateway evaluates the resource policy attached to your API, the result is affected by the authentication type that you have defined for the API, as illustrated in the flowcharts in the following sections.

Topics

API Gateway resource policy only

In this workflow, an API Gateway resource policy is attached to the API, but no authentication type is defined for the API. Evaluation of the policy involves seeking an explicit allow based on the inbound criteria of the caller. An implicit denial or any explicit denial results in denying the caller.

The following is an example of such a resource policy.

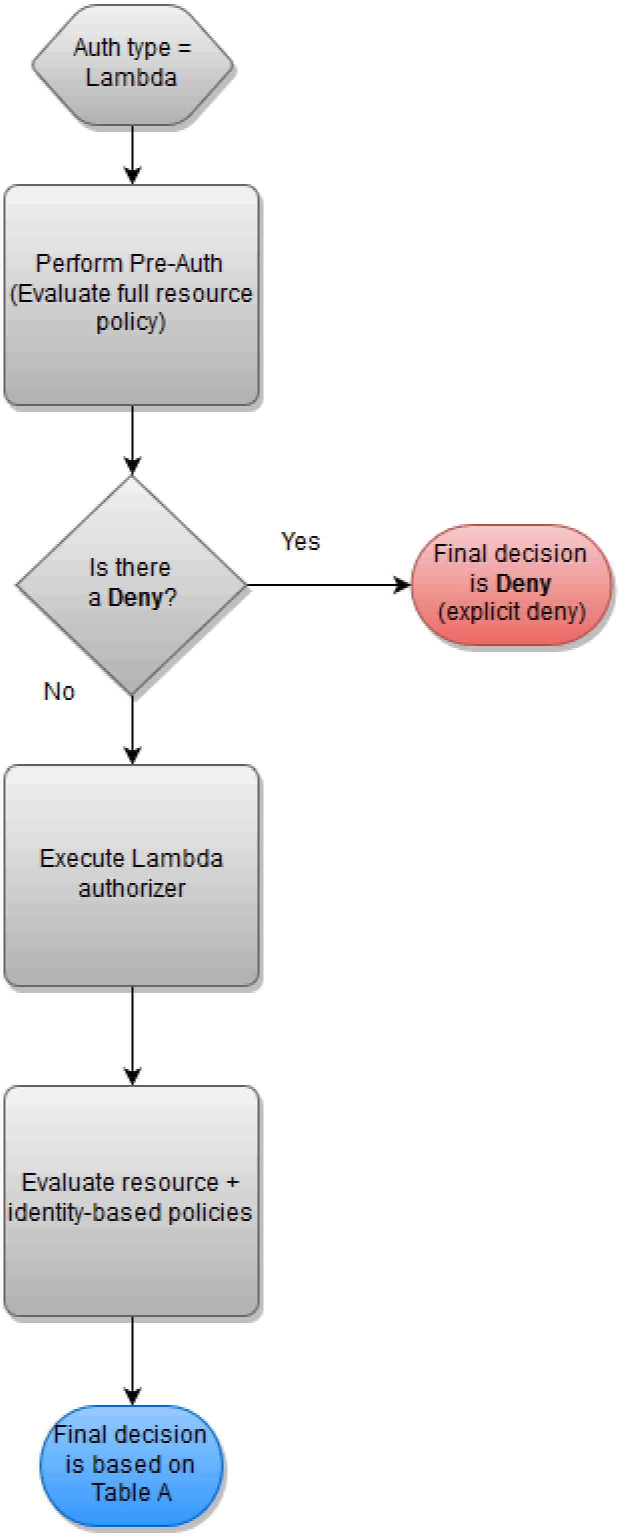

Auth type = None
Is there a resource policy?
No
Final decision is Allow (allow invoke)
Yes
Is there an explicit Allow?
Yes
No
Final decision is Deny (deny invoke)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:api-id/",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                }
            }
        }
    ]
}
```

Lambda authorizer and resource policy

In this workflow, a Lambda authorizer is configured for the API in addition to a resource policy. The resource policy is evaluated in two phases. Before calling the Lambda authorizer, API Gateway first evaluates the policy and checks for any explicit denials. If found, the caller is denied access immediately. Otherwise, the Lambda authorizer is called, and it returns a policy document, which is evaluated in conjunction with the resource policy. The result is determined based on Table A (near the end of this topic).

The following example resource policy allows calls only from the VPC endpoint whose VPC endpoint ID is *vpce-1a2b3c4d*. During the "pre-auth" evaluation, only the calls coming from the VPC endpoint indicated in the example are allowed to move forward and evaluate the Lambda authorizer. All remaining calls are blocked.

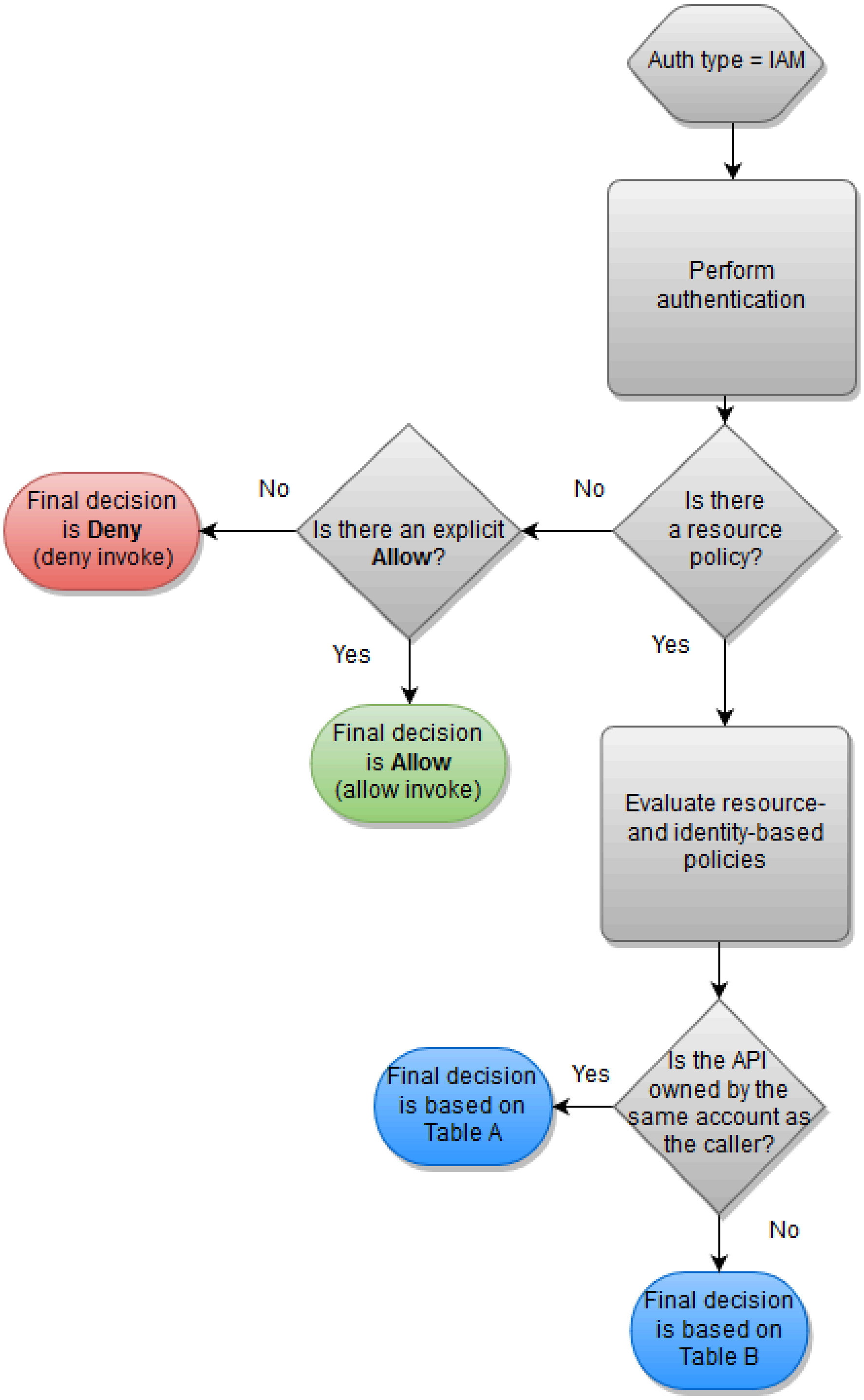

Auth type = Lambda
Perform Pre-Auth (Evaluate full resource policy)
Is there a **Deny**?
Yes
Final decision is **Deny** (explicit deny)
No
Execute Lambda authorizer
Evaluate resource + identity-based policies
Final decision is based on Table A

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "arn:aws:execute-api:region:account-id:api-id/"
            ],
            "Condition" : {
                "StringNotEquals": {
                    "aws:SourceVpce": "vpce-1a2b3c4d"
                }
            }
        }
    ]
}
```

IAM authentication and resource policy

In this workflow, you configure IAM authentication for the API in addition to a resource policy. After you authenticate the user with the IAM service, the API evaluates both the policies attached to the user and the resource policy. The outcome varies based on whether the caller is in the same AWS account or a separate AWS account, from the API owner.

If the caller and API owner are from separate accounts, both the IAM policies and the resource policy explicitly allow the caller to proceed. (See Table B at the end of this topic.) However, if the caller and the API owner are in the same AWS account, then either the IAM user policies or the resource policy must explicitly allow the caller to proceed. (See Table A below.)

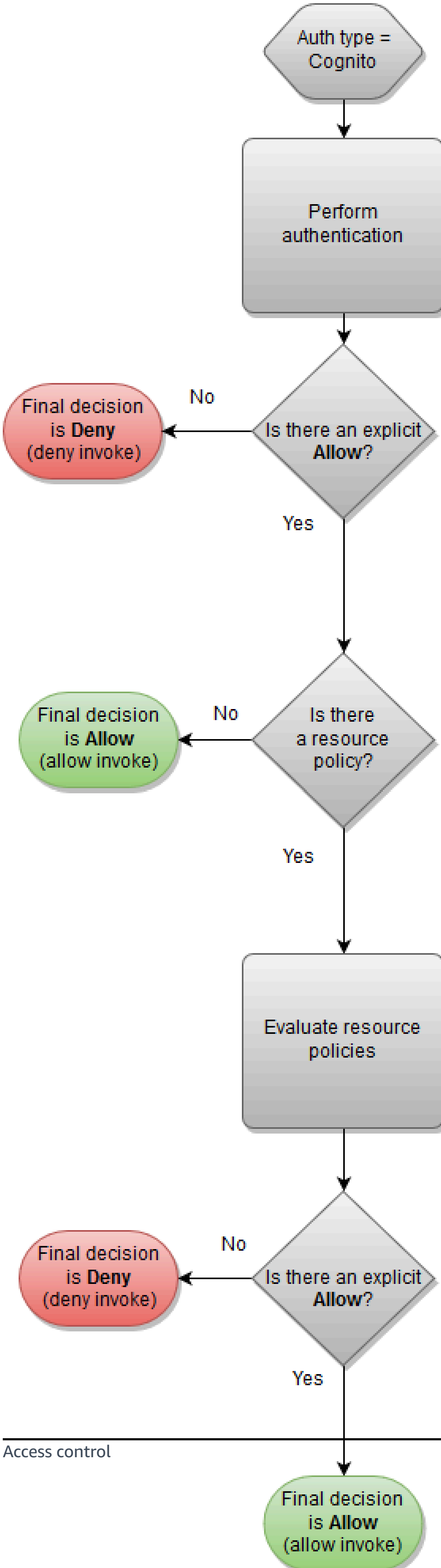

Auth type = IAM
Perform authentication
Is there a resource policy?
No
Is there an explicit **Allow**?
No
Final decision is **Deny** (deny invoke)
Yes
Final decision is **Allow** (allow invoke)
Yes
Evaluate resource- and identity-based policies
Is the API owned by the same account as the caller?
Yes
Final decision is based on Table A
No
Final decision is based on Table B

The following is an example of a cross-account resource policy. Assuming the IAM policy contains an allow effect, this resource policy allows calls only from the VPC whose VPC ID is *vpc-2f09a348*. (See Table B at the end of this topic.)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "arn:aws:execute-api:region:account-id:api-id/"
            ],
            "Condition" : {
                "StringEquals": {
                    "aws:SourceVpc": "vpc-2f09a348"
                    }
            }
        }
    ]
}
```

Amazon Cognito authentication and resource policy

In this workflow, an Amazon Cognito user pool is configured for the API in addition to a resource policy. API Gateway first attempts to authenticate the caller through Amazon Cognito. This is typically performed through a JWT token that is provided by the caller. If authentication is successful, the resource policy is evaluated independently, and an explicit allow is required. A deny or "neither allow or deny" results in a deny. The following is an example of a resource policy that might be used together with Amazon Cognito user pools.


Auth type = Cognito
Perform authentication
Is there an explicit **Allow**?
No
Final decision is **Deny** (deny invoke)
Yes
Is there a resource policy?
No
Final decision is **Allow** (allow invoke)
Yes
Evaluate resource policies
Is there an explicit **Allow**?
No
Final decision is **Deny** (deny invoke)
Yes
Final decision is **Allow** (allow invoke)

The following is an example of a resource policy that allows calls only from specified source IPs, assuming that the Amazon Cognito authentication token contains an allow. (See Table A near the end of this topic.)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:api-id/",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                }
            }
        }
    ]
}
```

Policy evaluation outcome tables

Table A lists the resulting behavior when access to an API Gateway API is controlled by an IAM policy (or a Lambda or Amazon Cognito user pools authorizer) and an API Gateway resource policy, both of which are in the same AWS account.

Table A: Account A Calls API Owned by Account A

IAM policy (or Lambda or Amazon Cognito user pools authorizer)	API Gateway resource policy	Resulting behavior
Allow	Allow	Allow
Allow	Neither Allow nor Deny	Allow
Allow	Deny	Explicit Deny
Neither Allow nor Deny	Allow	Allow
Neither Allow nor Deny	Neither Allow nor Deny	Implicit Deny

IAM policy (or Lambda or Amazon Cognito user pools authorizer)	API Gateway resource policy	Resulting behavior
Neither Allow nor Deny	Deny	Explicit Deny
Deny	Allow	Explicit Deny
Deny	Neither Allow nor Deny	Explicit Deny
Deny	Deny	Explicit Deny

Table B lists the resulting behavior when access to an API Gateway API is controlled by an IAM policy (or a Lambda or Amazon Cognito user pools authorizer) and an API Gateway resource policy, which are in different AWS accounts. If either is silent (neither allow nor deny), cross-account access is denied. This is because cross-account access requires that both the resource policy and the IAM policy (or a Lambda or Amazon Cognito user pools authorizer) explicitly grant access.

Table B: Account B Calls API Owned by Account A

IAM policy (or Lambda or Amazon Cognito user pools authorizer)	API Gateway resource policy	Resulting behavior
Allow	Allow	Allow
Allow	Neither Allow nor Deny	Implicit Deny
Allow	Deny	Explicit Deny
Neither Allow nor Deny	Allow	Implicit Deny
Neither Allow nor Deny	Neither Allow nor Deny	Implicit Deny
Neither Allow nor Deny	Deny	Explicit Deny
Deny	Allow	Explicit Deny
Deny	Neither Allow nor Deny	Explicit Deny

IAM policy (or Lambda or Amazon Cognito user pools authorizer)	API Gateway resource policy	Resulting behavior
Deny	Deny	Explicit Deny

API Gateway resource policy examples

This page presents a few examples of typical use cases for API Gateway resource policies.

The following example policies use a simplified syntax to specify the API resource. This simplified syntax is an abbreviated way that you can refer to an API resource, instead of specifying the full Amazon Resource Name (ARN). API Gateway converts the abbreviated syntax to the full ARN when you save the policy. For example, you can specify the resource `execute-api:/`*`stage-name`*`/`*`GET`*`/`*`pets`* in a resource policy. API Gateway converts the resource to `arn:aws:execute-api:us-east-2:123456789012:aabbccddee/stage-name/GET/pets` when you save the resource policy. API Gateway builds the full ARN by using the current Region, your AWS account ID, and the ID of the REST API that the resource policy is associated with. You can use `execute-api:/*` to represent all stages, methods, and paths in the current API. For information about access policy language, see Access policy language overview for Amazon API Gateway.

Topics

- Example: Allow roles in another AWS account to use an API
- Example: Deny API traffic based on source IP address or range
- Example: Deny API traffic based on source IP address or range when using a private API
- Example: Allow private API traffic based on source VPC or VPC endpoint

Example: Allow roles in another AWS account to use an API

The following example resource policy grants API access in one AWS account to two roles in a different AWS account via Signature Version 4 (SigV4) protocols. Specifically, the developer and the administrator role for the AWS account identified by *`account-id-2`* are granted the `execute-api:Invoke` action to execute the `GET` action on the `pets` resource (API) in your AWS account.

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::account-id-2:role/developer",
                    "arn:aws:iam::account-id-2:role/Admin"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/stage/GET/pets"
            ]
        }
    ]
}
```

Example: Deny API traffic based on source IP address or range

The following example resource policy denies (blocks) incoming traffic to an API from two specified source IP address blocks.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                }
            }
        }
    ]
}
```

Example: Deny API traffic based on source IP address or range when using a private API

The following example resource policy denies (blocks) incoming traffic to a private API from two specified source IP address blocks. When using private APIs, the VPC endpoint for `execute-api` re-writes the original source IP address. The `aws:VpcSourceIp` condition filters the request against the original requester IP address.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:VpcSourceIp": ["192.0.2.0/24", "198.51.100.0/24"]
                }
            }
        }
    ]
}
```

Example: Allow private API traffic based on source VPC or VPC endpoint

The following example resource policies allow incoming traffic to a private API only from a specified virtual private cloud (VPC) or VPC endpoint.

This example resource policy specifies a source VPC:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ],
            "Condition" : {
                "StringNotEquals": {
                   "aws:SourceVpc": "vpc-1a2b3c4d"
                }
            }
        }
    ]
}
```

This example resource policy specifies a source VPC endpoint:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ],
            "Condition" : {
                "StringNotEquals": {
                    "aws:SourceVpce": "vpce-1a2b3c4d"
                }
            }
        }
    ]
}
```

Create and attach an API Gateway resource policy to an API

To allow a user to access your API by calling the API execution service, you must create an API Gateway resource policy, which controls access to the API Gateway resources, and attach the policy to the API.

> ⚠ **Important**
>
> To update an API Gateway resource policy, you'll need to have `apigateway:UpdateRestApiPolicy` permission in addition to `apigateway:PATCH` permission.

The resource policy can be attached to the API when the API is being created, or it can be attached afterwards. For private APIs, note that until you attach the resource policy to the private API, all calls to the API will fail.

> ⚠ **Important**
>
> If you update the resource policy after the API is created, you'll need to deploy the API to propagate the changes after you've attached the updated policy. Updating or saving the

policy alone won't change the runtime behavior of the API. For more information about deploying your API, see Deploying a REST API in Amazon API Gateway.

You can control access by IAM condition elements, including conditions on AWS accounts, source VPCs, source VPC endpoints, or IP ranges. If you set the `Principal` in the policy to `"*"`, you can use other authorization types alongside the resource policy.

However, if you set the `Principal` to an AWS principal, such as the following: `"Principal": { "AWS": "arn:aws:iam..." }` Authorization fails for all resources not secured with `AWS_IAM` authorization, including unsecured resources.

The following sections describe how to create your own API Gateway resource policy and attach it to your API. Attaching a policy applies the permissions in the policy to the methods in the API.

⚠ **Important**

If you use the API Gateway console to attach a resource policy to a deployed API, or if you update an existing resource policy, you'll need to redeploy the API in the console for the changes to take effect.

Topics

- Attaching API Gateway resource policies (console)
- Attaching API Gateway resource policies (AWS CLI)
- Attaching API Gateway resource policies (AWS CloudFormation)

Attaching API Gateway resource policies (console)

You can use the AWS Management console to attach a resource policy to an API Gateway API.

To attach a resource policy to an API Gateway API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **Resource policy**.
4. Choose **Create policy**.

5. (Optional) Choose **Select a template** to generate an example policy.

   In the example policies, placeholders are enclosed in double curly braces ("{{*placeholder*}}"). Replace each of the placeholders, including the curly braces, with the necessary information.
6. If you don't use one of the template examples, enter your resource policy.
7. Choose **Save changes**.

If the API has been deployed previously in the API Gateway console, you'll need to redeploy it for the resource policy to take effect.

Attaching API Gateway resource policies (AWS CLI)

To use the AWS CLI to create a new API and attach a resource policy to it, call the `create-rest-api` command as follows:

```
aws apigateway create-rest-api \
    --name "api-name" \
    --policy "{\"jsonEscapedPolicyDocument\"}"
```

To use the AWS CLI to attach a resource policy to an existing API, call the `update-rest-api` command as follows:

```
aws apigateway update-rest-api \
    --rest-api-id api-id \
    --patch-operations op=replace,path=/
policy,value='"{\"jsonEscapedPolicyDocument\"}"'
```

Attaching API Gateway resource policies (AWS CloudFormation)

You can use AWS CloudFormation to create an API with a resource policy. The following example creates a REST API with the example resource policy, the section called "Example: Deny API traffic based on source IP address or range".

Example AWS CloudFormation template

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: testapi
      Policy:
        Statement:
          - Action: 'execute-api:Invoke'
            Effect: Allow
            Principal: '*'
            Resource: 'execute-api/*'
          - Action: 'execute-api:Invoke'
            Effect: Deny
            Principal: '*'
            Resource: 'execute-api/*'
            Condition:
              IpAddress:
                'aws:SourceIp': ["192.0.2.0/24", "198.51.100.0/24" ]
        Version: 2012-10-17
  Resource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'helloworld'
  MethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref Resource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: MOCK
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - MethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: test
```

AWS condition keys that can be used in API Gateway resource policies

The following table contains AWS condition keys that can be used in resource policies for APIs in API Gateway for each authorization type.

For more information about AWS condition keys, see AWS Global Condition Context Keys.

Table of condition keys

Condition keys	Criteria	Needs AuthN?	Authorization type
`aws:CurrentTime`	None	No	All
`aws:EpochTime`	None	No	All
`aws:TokenIssueTime`	Key is present only in requests that are signed using temporary security credentials.	Yes	IAM
`aws:MultiFactorAuthPresent`	Key is present only in requests that are signed using temporary security credentials.	Yes	IAM
`aws:MultiFactorAuthAge`	Key is present only if MFA is present in the requests.	Yes	IAM
`aws:PrincipalAccount`	None	Yes	IAM
`aws:PrincipalArn`	None	Yes	IAM
`aws:PrincipalOrgID`	This key is included in the request context only if the principal	Yes	IAM

Condition keys	Criteria	Needs AuthN?	Authorization type
	is a member of an organization.		
`aws:PrincipalOrgPaths`	This key is included in the request context only if the principal is a member of an organization.	Yes	IAM
`aws:PrincipalTag`	This key is included in the request context if the principal is using an IAM user with attached tags. It is included for a principal using an IAM role with attached tags or session tags.	Yes	IAM
`aws:PrincipalType`	None	Yes	IAM
`aws:Referer`	Key is present only if the value is provided by the caller in the HTTP header.	No	All
`aws:SecureTransport`	None	No	All
`aws:SourceArn`	None	No	All
`aws:SourceIp`	None	No	All
`aws:SourceVpc`	This key can be used only for private APIs.	No	All

Condition keys	Criteria	Needs AuthN?	Authorization type
`aws:SourceVpce`	This key can be used only for private APIs.	No	All
`aws:UserAgent`	Key is present only if the value is provided by the caller in the HTTP header.	No	All
`aws:userid`	None	Yes	IAM
`aws:username`	None	Yes	IAM

Control access to an API with IAM permissions

You control access to your Amazon API Gateway API with IAM permissions by controlling access to the following two API Gateway component processes:

- To create, deploy, and manage an API in API Gateway, you must grant the API developer permissions to perform the required actions supported by the API management component of API Gateway.
- To call a deployed API or to refresh the API caching, you must grant the API caller permissions to perform required IAM actions supported by the API execution component of API Gateway.

The access control for the two processes involves different permissions models, explained next.

API Gateway permissions model for creating and managing an API

To allow an API developer to create and manage an API in API Gateway, you must create IAM permissions policies that allow a specified API developer to create, update, deploy, view, or delete required API entities. You attach the permissions policy to a user, role, or group.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Creating a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Creating a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

For more information on how to use this permissions model, see the section called “API Gateway identity-based policies”.

API Gateway permissions model for invoking an API

To allow an API caller to invoke the API or refresh its caching, you must create IAM policies that permit a specified API caller to invoke the API method for which user authentication is enabled. The API developer sets the method's `authorizationType` property to `AWS_IAM` to require that the caller submit the user's credentials to be authenticated. Then, you attach the policy to a user, role, or group.

In this IAM permissions policy statement, the IAM `Resource` element contains a list of deployed API methods identified by given HTTP verbs and API Gateway resource paths. The IAM `Action` element contains the required API Gateway API executing actions. These actions include `execute-api:Invoke` or `execute-api:InvalidateCache`, where `execute-api` designates the underlying API execution component of API Gateway.

For more information on how to use this permissions model, see Control access for invoking an API.

When an API is integrated with an AWS service (for example, AWS Lambda) in the back end, API Gateway must also have permissions to access integrated AWS resources (for example, invoking a Lambda function) on behalf of the API caller. To grant these permissions, create an IAM role of the **AWS service for API Gateway** type. When you create this role in the IAM Management console, this resulting role contains the following IAM trust policy that declares API Gateway as a trusted entity permitted to assume the role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

If you create the IAM role by calling the create-role command of CLI or a corresponding SDK method, you must supply the above trust policy as the input parameter of `assume-role-policy-document`. Do not attempt to create such a policy directly in the IAM Management console or calling AWS CLI create-policy command or a corresponding SDK method.

For API Gateway to call the integrated AWS service, you must also attach to this role appropriate IAM permissions policies for calling integrated AWS services. For example, to call a Lambda function, you must include the following IAM permissions policy in the IAM role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "*"
        }
    ]
}
```

Note that Lambda supports resource-based access policy, which combines both trust and permissions policies. When integrating an API with a Lambda function using the API Gateway console, you are not asked to set this IAM role explicitly, because the console sets the resource-based permissions on the Lambda function for you, with your consent.

Note

To enact access control to an AWS service, you can use either the caller-based permissions model, where a permissions policy is directly attached to the caller's user or group, or

> the role-based permission model, where a permissions policy is attached to an IAM role that API Gateway can assume. The permissions policies may differ in the two models. For example, the caller-based policy blocks the access while the role-based policy allows it. You can take advantage of this to require that a user access an AWS service through an API Gateway API only.

Control access for invoking an API

In this section you will learn how to write up IAM policy statements to control who can call a deployed API in API Gateway. Here, you will also find the policy statement reference, including the formats of `Action` and `Resource` fields related to the API execution service. You should also study the IAM section in the section called "How resource policies affect authorization workflow".

For private APIs, you should use a combination of an API Gateway resource policy and a VPC endpoint policy. For more information, see the following topics:

- the section called "Use API Gateway resource policies"
- the section called "Use VPC endpoint policies for private APIs"

Control who can call an API Gateway API method with IAM policies

To control who can or cannot call a deployed API with IAM permissions, create an IAM policy document with required permissions. A template for such a policy document is shown as follows.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Permission",
      "Action": [
        "execute-api:Execution-operation"
      ],
      "Resource": [
        "arn:aws:execute-api:region:account-id:api-id/stage/METHOD_HTTP_VERB/Resource-
path"
      ]
    }
  ]
}
```

Here, *Permission* is to be replaced by Allow or Deny depending on whether you want to grant or revoke the included permissions. *Execution-operation* is to be replaced by the operations supported by the API execution service. *METHOD_HTTP_VERB* stands for a HTTP verb supported by the specified resources. *Resource-path* is the placeholder for the URL path of a deployed API Resource instance supporting the said *METHOD_HTTP_VERB*. For more information, see Statement reference of IAM policies for executing API in API Gateway.

Note

For IAM policies to be effective, you must have enabled IAM authentication on API methods by setting AWS_IAM for the methods' authorizationType property. Failing to do so will make these API methods publicly accessible.

For example, to grant a user permission to view a list of pets exposed by a specified API, but to deny the user permission to add a pet to the list, you could include the following statement in the IAM policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*/GET/pets"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*/POST/pets"
      ]
    }
  ]
}
```

To grant a user permission to view a specific pet exposed by an API that is configured as `GET /pets/{petId}`, you could include the following statement in the IAM policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*/GET/pets/a1b2"
      ]
    }
  ]
}
```

Statement reference of IAM policies for executing API in API Gateway

The following information describes the Action and Resource format of IAM policy statements of access permissions for executing an API.

Action format of permissions for executing API in API Gateway

The API-executing `Action` expression has the following general format:

```
execute-api:action
```

where *`action`* is an available API-executing action:

- *****, which represents all of the following actions.
- **Invoke**, used to invoke an API upon a client request.
- **InvalidateCache**, used to invalidate API cache upon a client request.

Resource format of permissions for executing API in API Gateway

The API-executing `Resource` expression has the following general format:

```
arn:aws:execute-api:region:account-id:api-id/stage-name/HTTP-VERB/resource-path-specifier
```

where:

- *region* is the AWS region (such as **us-east-1** or ***** for all AWS regions) that corresponds to the deployed API for the method.
- *account-id* is the 12-digit AWS account Id of the REST API owner.
- *api-id* is the identifier API Gateway has assigned to the API for the method.
- *stage-name* is the name of the stage associated with the method.
- *HTTP-VERB* is the HTTP verb for the method. It can be one of the following: GET, POST, PUT, DELETE, PATCH.
- *resource-path-specifier* is the path to the desired method.

> **Note**
>
> If you specify a wildcard (*), the Resource expression applies the wildcard to the rest of the expression.

Some example resource expressions include:

- **arn:aws:execute-api:*:*:*** for any resource path in any stage, for any API in any AWS region.
- **arn:aws:execute-api:us-east-1:*:*** for any resource path in any stage, for any API in the AWS region of us-east-1.
- **arn:aws:execute-api:us-east-1:*:*api-id*/*** for any resource path in any stage, for the API with the identifier of *api-id* in the AWS region of us-east-1.
- **arn:aws:execute-api:us-east-1:*:*api-id*/test/*** for resource path in the stage of test, for the API with the identifier of *api-id* in the AWS region of us-east-1.

To learn more, see API Gateway Amazon Resource Name (ARN) reference.

IAM policy examples for API execution permissions

For permissions model and other background information, see Control access for invoking an API.

The following policy statement gives the user permission to call any POST method along the path of `mydemoresource`, in the stage of `test`, for the API with the identifier of `a123456789`, assuming the corresponding API has been deployed to the AWS region of us-east-1:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:*:a123456789/test/POST/mydemoresource/*"
      ]
    }
  ]
}
```

The following example policy statement gives the user permission to call any method on the resource path of `petstorewalkthrough/pets`, in any stage, for the API with the identifier of `a123456789`, in any AWS region where the corresponding API has been deployed:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:*:*:a123456789/*/*/petstorewalkthrough/pets"
      ]
    }
  ]
}
```

Create and attach a policy to a user

To enable a user to call the API managing service or the API execution service, you must create an IAM policy which controls access to the API Gateway entities.

To use the JSON policy editor to create a policy

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Policies**.

   If this is your first time choosing **Policies**, the **Welcome to Managed Policies** page appears. Choose **Get Started**.
3. At the top of the page, choose **Create policy**.
4. In the **Policy editor** section, choose the **JSON** option.
5. Enter the following JSON policy document:

```
{
  "Version": "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    },
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    }
  ]
}
```

6. Choose **Next**.

> **Note**
>
> You can switch between the **Visual** and **JSON** editor options anytime. However, if you make changes or choose **Next** in the **Visual** editor, IAM might restructure your policy to optimize it for the visual editor. For more information, see Policy restructuring in the *IAM User Guide*.

7. On the **Review and create** page, enter a **Policy name** and a **Description** (optional) for the policy that you are creating. Review **Permissions defined in this policy** to see the permissions that are granted by your policy.
8. Choose **Create policy** to save your new policy.

In this statement, substitute `action-statement` and `resource-statement` as needed, and add other statements to specify the API Gateway entities you want to allow the user to manage, the API methods the user can call, or both. By default, the user does not have permissions unless there is an explicit corresponding `Allow` statement.

You have just created an IAM policy. It won't have any effect until you attach it.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Creating a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Creating a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

To attach an IAM policy document to an IAM group

1. Choose **Groups** from the main navigation pane.
2. Choose the **Permissions** tab under the chosen group.
3. Choose **Attach policy**.
4. Choose the policy document that you previously created, and then choose **Attach policy**.

For API Gateway to call other AWS services on your behalf, create an IAM role of the **Amazon API Gateway** type.

To create an Amazon API Gateway type of role

1. Choose **Roles** from the main navigation pane.
2. Choose **Create New Role**.
3. Type a name for **Role name** and then choose **Next Step**.
4. Under **Select Role Type**, in **AWS Service Roles**, choose **Select** next to **Amazon API Gateway**.
5. Choose an available managed IAM permissions policy, for example, **AmazonAPIGatewayPushToCloudWatchLog** if you want API Gateway to log metrics in CloudWatch, under **Attach Policy** and then choose **Next Step**.
6. Under **Trusted Entities**, verify that **apigateway.amazonaws.com** is listed as an entry, and then choose **Create Role**.
7. In the newly created role, choose the **Permissions** tab and then choose **Attach Policy**.
8. Choose the previously created custom IAM policy document and then choose **Attach Policy**.

Use VPC endpoint policies for private APIs in API Gateway

You can improve the security of your private APIs by configuring API Gateway to use an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access AWS services by using private IP addresses. For more information about creating VPC endpoints, see Creating an Interface Endpoint.

A *VPC endpoint policy* is an IAM resource policy that you can attach to an interface VPC endpoint to control access to the endpoint. For more information, see Controlling Access to Services with VPC Endpoints. You can use an endpoint policy to restrict the traffic going from your internal network to access your private APIs. You can choose to allow or disallow access to specific private APIs that can be accessed through the VPC endpoint.

VPC endpoint policies can be used together with API Gateway resource policies. The resource policy is used to specify which principals can access the API. The endpoint policy specifies which private APIs can be called via the VPC endpoint. For more information about resource policies, see the section called "Use API Gateway resource policies".

VPC endpoint policy considerations

- If a policy restricts IAM principals, you must set the `authorizationType` of the method to `AWS_IAM` or `NONE`.
- The identity of the invoker is evaluated based on the `Authorization` header value. Depending on your `authorizationType`, this may lead to an `403 IncompleteSignatureException` or an `403 InvalidSignatureException` error. The following table shows the `Authorization` header values for each `authorizationType`.

`authorizationType`	**`Authorization` header evaluated?**	**Allowed `Authorization` header values**
`NONE` with the default full access policy	No	Not passed
`NONE` with a custom access policy	Yes	Must be a valid SigV4 value
`IAM`	Yes	Must be a valid SigV4 value
`CUSTOM` or `COGNITO_USER_POOLS`	No	Not passed

VPC endpoint policy examples

You can create policies for Amazon Virtual Private Cloud endpoints for Amazon API Gateway in which you can specify:

- The principal that can perform actions.
- The actions that can be performed.
- The resources that can have actions performed on them.

To attach the policy to the VPC endpoint, you'll need to use the VPC console. For more information, see Controlling Access to Services with VPC Endpoints.

Example 1: VPC endpoint policy granting access to two APIs

The following example policy grants access to only two specific APIs via the VPC endpoint that the policy is attached to.

```
{
    "Statement": [
        {
            "Principal": "*",
            "Action": [
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/*",
                "arn:aws:execute-api:us-east-1:123412341234:aaaaa11111/*"
            ]
        }
    ]
}
```

Example 2: VPC endpoint policy granting access to GET methods

The following example policy grants users access to GET methods for a specific API via the VPC endpoint that the policy is attached to.

```
{
    "Statement": [
        {
            "Principal": "*",
            "Action": [
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/stageName/GET/*"
            ]
        }
    ]
}
```

Example 3: VPC endpoint policy granting a specific user access to a specific API

The following example policy grants a specific user access to a specific API via the VPC endpoint that the policy is attached to.

```
{
    "Statement": [
        {
            "Principal": {
                "AWS": [
                    "arn:aws:iam::123412341234:user/MyUser"
                ]
            },
            "Action": [
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/*"
            ]
        }
    ]
}
```

Using tags to control access to a REST API in API Gateway

Permission to access REST APIs can be fine-tuned using attribute-based access control in IAM policies.

For more information, see the section called “Attribute-based access control”.

Use API Gateway Lambda authorizers

A *Lambda authorizer* (formerly known as a *custom authorizer*) is an API Gateway feature that uses a Lambda function to control access to your API.

A Lambda authorizer is useful if you want to implement a custom authorization scheme that uses a bearer token authentication strategy such as OAuth or SAML, or that uses request parameters to determine the caller's identity.

When a client makes a request to one of your API's methods, API Gateway calls your Lambda authorizer, which takes the caller's identity as input and returns an IAM policy as output.

There are two types of Lambda authorizers:

- A *token-based* Lambda authorizer (also called a `TOKEN` authorizer) receives the caller's identity in a bearer token, such as a JSON Web Token (JWT) or an OAuth token. For an example application, see Open Banking Brazil - Authorization Samples on GitHub.
- A *request parameter-based* Lambda authorizer (also called a `REQUEST` authorizer) receives the caller's identity in a combination of headers, query string parameters, `stageVariables`, and `$context` variables.

  For WebSocket APIs, only request parameter-based authorizers are supported.

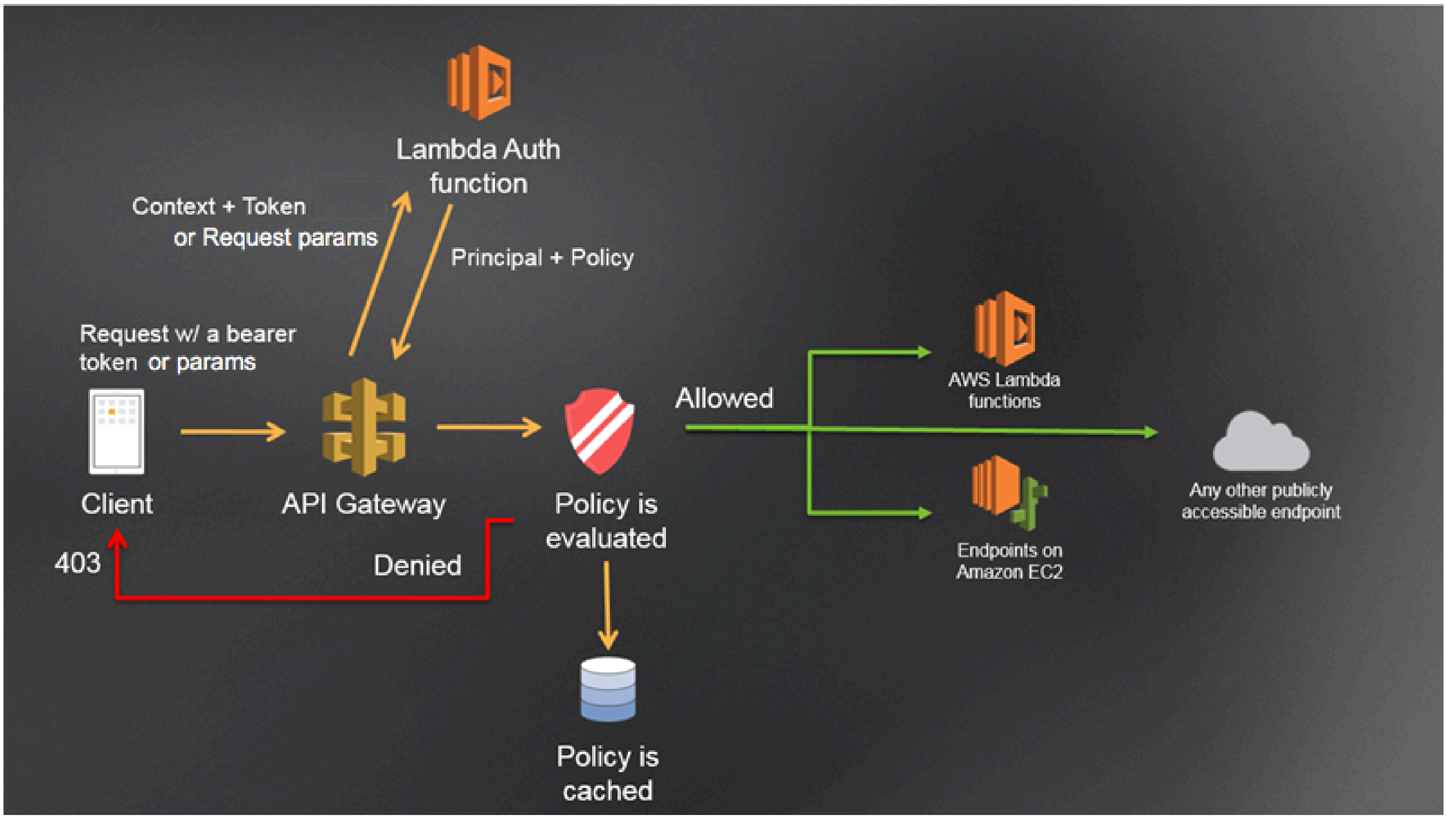
It is possible to use an AWS Lambda function from an AWS account that is different from the one in which you created your API. For more information, see the section called “Configure a cross-account Lambda authorizer”.

For example Lambda functions, see aws-apigateway-lambda-authorizer-blueprints on GitHub.

Topics

Lambda authorizer Auth workflow

The following diagram illustrates the authorization workflow for Lambda authorizers.



API Gateway Lambda authorization workflow

1. The client calls a method on an API Gateway API method, passing a bearer token or request parameters.
2. API Gateway checks whether a Lambda authorizer is configured for the method. If it is, API Gateway calls the Lambda function.
3. The Lambda function authenticates the caller by means such as the following:
   - Calling out to an OAuth provider to get an OAuth access token.
   - Calling out to a SAML provider to get a SAML assertion.
   - Generating an IAM policy based on the request parameter values.
   - Retrieving credentials from a database.
4. If the call succeeds, the Lambda function grants access by returning an output object containing at least an IAM policy and a principal identifier.
5. API Gateway evaluates the policy.
   - If access is denied, API Gateway returns a suitable HTTP status code, such as `403 ACCESS_DENIED`.

- If access is allowed, API Gateway invokes the method. If caching is enabled in the authorizer settings, API Gateway also caches the policy so that the Lambda authorizer function doesn't need to be invoked again.

6. The call can fail if the Lambda function returns a `401 Unauthorized` response. You can customize the `401 Unauthorized` gateway response. To learn more, see the section called "Gateway responses".

Steps to create an API Gateway Lambda authorizer

To create a Lambda authorizer, you need to perform the following tasks:

1. Create the Lambda authorizer function in the Lambda console as described in the section called "Create a Lambda authorizer function in the Lambda console". You can use one of the blueprint examples as a starting point and customize the input and output as desired.
2. Configure the Lambda function as an API Gateway authorizer and configure an API method to require it, as described in the section called "Configure a Lambda authorizer using the console". Alternatively, if you need a cross-account Lambda authorizer, see the section called "Configure a cross-account Lambda authorizer".

   > **Note**
   >
   > You can also configure an authorizer by using the AWS CLI or an AWS SDK.

3. Test your authorizer by using Postman as described in the section called "Call an API with Lambda authorizers".

Create an API Gateway Lambda authorizer function in the Lambda console

Before configuring a Lambda authorizer, you must first create the Lambda function that implements the logic to authorize and, if necessary, to authenticate the caller. The Lambda console provides a Python blueprint, which you can use by choosing **Use a blueprint** and choosing the **api-gateway-authorizer-python** blueprint. Otherwise, you'll want to use one of the blueprints in the awslabs GitHub repository as a starting point.

For the example Lambda authorizer functions in this section, which don't call other services, you can use the built-in `AWSLambdaBasicExecutionRole`. When creating the Lambda function for your own API Gateway Lambda authorizer, you'll need to assign an IAM execution role to the

Lambda function if it calls other AWS services. To create the role, follow the instructions in AWS Lambda Execution Role.

For more example Lambda functions, see aws-apigateway-lambda-authorizer-blueprints on GitHub. For an example application, see Open Banking Brazil - Authorization Samples on GitHub.

EXAMPLE: Create a token-based Lambda authorizer function

To create a token-based Lambda authorizer function, enter the following Node.js code for the most recent runtime in the Lambda console. Then, you test the authorizer in the API Gateway console.

To create the token-based Lambda authorizer function

1. In the Lambda console, choose **Create function**.
2. Choose **Author from scratch**.
3. Enter a name for the function.
4. For **Runtime**, choose either the latest supported **Node.js** or **Python** runtime.
5. Choose **Create function**.
6. Copy/paste the following code into the code editor.

   Node.js

   ```
   // A simple token-based authorizer example to demonstrate how to use an
    authorization token
   // to allow or deny a request. In this example, the caller named 'user' is
    allowed to invoke
   // a request if the client-supplied token value is 'allow'. The caller is not
    allowed to invoke
   // the request if the token value is 'deny'. If the token value is
    'unauthorized' or an empty
   // string, the authorizer function returns an HTTP 401 status code. For any
    other token value,
   // the authorizer returns an HTTP 500 status code.
   // Note that token values are case-sensitive.

   export const handler =  function(event, context, callback) {
       var token = event.authorizationToken;
       switch (token) {
           case 'allow':
               callback(null, generatePolicy('user', 'Allow', event.methodArn));
               break;
   ```

```
        case 'deny':
            callback(null, generatePolicy('user', 'Deny', event.methodArn));
            break;
        case 'unauthorized':
            callback("Unauthorized");   // Return a 401 Unauthorized response
            break;
        default:
            callback("Error: Invalid token"); // Return a 500 Invalid token
 response
    }
};

// Help function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
    var authResponse = {};

    authResponse.principalId = principalId;
    if (effect && resource) {
        var policyDocument = {};
        policyDocument.Version = '2012-10-17';
        policyDocument.Statement = [];
        var statementOne = {};
        statementOne.Action = 'execute-api:Invoke';
        statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }

    // Optional output with custom properties of the String, Number or Boolean
 type.
    authResponse.context = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": true
    };
    return authResponse;
}
```

Python

```
# A simple token-based authorizer example to demonstrate how to use an
 authorization token
```

```
# to allow or deny a request. In this example, the caller named 'user' is
 allowed to invoke
# a request if the client-supplied token value is 'allow'. The caller is not
 allowed to invoke
# the request if the token value is 'deny'. If the token value is 'unauthorized'
 or an empty
# string, the authorizer function returns an HTTP 401 status code. For any other
 token value,
# the authorizer returns an HTTP 500 status code.
# Note that token values are case-sensitive.

import json


def lambda_handler(event, context):
    token = event['authorizationToken']
    if token == 'allow':
        print('authorized')
        response = generatePolicy('user', 'Allow', event['methodArn'])
    elif token == 'deny':
        print('unauthorized')
        response = generatePolicy('user', 'Deny', event['methodArn'])
    elif token == 'unauthorized':
        print('unauthorized')
        raise Exception('Unauthorized')  # Return a 401 Unauthorized response
        return 'unauthorized'
    try:
        return json.loads(response)
    except BaseException:
        print('unauthorized')
        return 'unauthorized'  # Return a 500 error


def generatePolicy(principalId, effect, resource):
    authResponse = {}
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument
    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }
    authResponse_JSON = json.dumps(authResponse)
    return authResponse_JSON
```

7. Choose **Deploy**.

After you create your Lambda function, you create and test a token-based Lambda authorizer in the API Gateway console.

To create a token-based Lambda authorizer

1. In the API Gateway console, create a simple API if you don't already have one.
2. Choose your API from the API list.
3. Choose **Authorizers**.
4. Choose **Create authorizer**.
5. For **Authorizer name**, enter a name.
6. For **Authorizer type**, select **Lambda**.
7. For **Lambda function**, select the AWS Region where you created your Lambda authorizer function, and then enter the function name.
8. Keep **Lambda invoke role** blank.
9. For **Lambda event payload**, select **Token**.
10. For **Token source**, enter **`authorizationToken`**.
11. Choose **Create authorizer**.

To test your authorizer

1. Select the name of the authorizer.
2. Under **Test authorizer**, for the **authorizationToken** value, enter **`allow`**.
3. Choose **Test authorizer**.

In this example, when the API receives a method request, API Gateway passes the source token to this Lambda authorizer function in the `event.authorizationToken` attribute. The Lambda authorizer function reads the token and acts as follows:

- If the token value is `'allow'`, the test of the authorizer function returns a `200 OK` HTTP response and an IAM policy that looks like the following, and the method request succeeds:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Allow",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

- If the token value is `'deny'`, the test of the authorizer function returns a `200 OK` HTTP response and a `Deny` IAM policy that looks like the following, and the method request fails:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Deny",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

Note

Outside of the test environment, the authorizer function returns a `403 Forbidden` HTTP response and the method request fails.

- If the token value is `'unauthorized'` or an empty string, the test of the authorizer function returns a `401 Unauthorized` HTTP response, and the method call fails.

- If the token is anything else, the client receives a `500 Invalid token` response, and the method call fails.

> **Note**
>
> In production code, you may need to authenticate the user before granting authorization. If so, you can add authentication logic in the Lambda function as well by calling an authentication provider as directed in the documentation for that provider.

In addition to returning an IAM policy, the Lambda authorizer function must also return the caller's principal identifier. It can also optionally return a `context` object containing additional information that can be passed into the integration backend. For more information, see [Output from an Amazon API Gateway Lambda authorizer](#).

EXAMPLE: Create a request-based Lambda authorizer function

To create a request-based Lambda authorizer function, enter the following Node.js code for the most recent runtime in the Lambda console. Then, you test the authorizer in the API Gateway console.

1. In the Lambda console, choose **Create function**.
2. Choose **Author from scratch**.
3. Enter a name for the function.
4. For **Runtime**, choose either the latest supported **Node.js** or **Python** runtime.
5. Choose **Create function**.
6. Copy/paste the following code into the code editor.

   Node.js

```
  // A simple request-based authorizer example to demonstrate how to use
request
   // parameters to allow or deny a request. In this example, a request is
   // authorized if the client-supplied headerauth1 header, QueryString1
   // query parameter, and stage variable of StageVar1 all match
   // specified values of 'headerValue1', 'queryValue1', and 'stageValue1',
   // respectively.
```

```
export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));

    // Retrieve request parameters from the Lambda function input:
    var headers = event.headers;
    var queryStringParameters = event.queryStringParameters;
    var pathParameters = event.pathParameters;
    var stageVariables = event.stageVariables;

    // Parse the input for the parameter values
    var tmp = event.methodArn.split(':');
    var apiGatewayArnTmp = tmp[5].split('/');
    var awsAccountId = tmp[4];
    var region = tmp[3];
    var restApiId = apiGatewayArnTmp[0];
    var stage = apiGatewayArnTmp[1];
    var method = apiGatewayArnTmp[2];
    var resource = '/'; // root resource
    if (apiGatewayArnTmp[3]) {
        resource += apiGatewayArnTmp[3];
    }

    // Perform authorization to return the Allow policy for correct parameters
 and
    // the 'Unauthorized' error, otherwise.
    var authResponse = {};
    var condition = {};
    condition.IpAddress = {};

    if (headers.headerauth1 === "headerValue1"
        && queryStringParameters.QueryString1 === "queryValue1"
        && stageVariables.StageVar1 === "stageValue1") {
        callback(null, generateAllow('me', event.methodArn));
    }  else {
        callback("Unauthorized");
    }
}

// Help function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
    // Required output:
    var authResponse = {};
    authResponse.principalId = principalId;
    if (effect && resource) {
        var policyDocument = {};
        policyDocument.Version = '2012-10-17'; // default version
        policyDocument.Statement = [];
        var statementOne = {};
        statementOne.Action = 'execute-api:Invoke'; // default action
        statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }
    // Optional output with custom properties of the String, Number or Boolean
 type.
    authResponse.context = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": true
    };
    return authResponse;
}

var generateAllow = function(principalId, resource) {
    return generatePolicy(principalId, 'Allow', resource);
}

var generateDeny = function(principalId, resource) {
    return generatePolicy(principalId, 'Deny', resource);
}
```

Python

```
# A simple request-based authorizer example to demonstrate how to use request
# parameters to allow or deny a request. In this example, a request is
# authorized if the client-supplied headerauth1 header, QueryString1
# query parameter, and stage variable of StageVar1 all match
# specified values of 'headerValue1', 'queryValue1', and 'stageValue1',
# respectively.

import json


def lambda_handler(event, context):
    print(event)
    # Retrieve request parameters from the Lambda function input:
    headers = event['headers']
    queryStringParameters = event['queryStringParameters']
    pathParameters = event['pathParameters']
    stageVariables = event['stageVariables']

    # Parse the input for the parameter values
    tmp = event['methodArn'].split(':')
    apiGatewayArnTmp = tmp[5].split('/')
    awsAccountId = tmp[4]
    region = tmp[3]
    restApiId = apiGatewayArnTmp[0]
    stage = apiGatewayArnTmp[1]
    method = apiGatewayArnTmp[2]
    resource = '/'

    if (apiGatewayArnTmp[3]):
        resource += apiGatewayArnTmp[3]

    # Perform authorization to return the Allow policy for correct parameters
    # and the 'Unauthorized' error, otherwise.

    authResponse = {}
    condition = {}
    condition['IpAddress'] = {}

    if (headers['headerauth1'] == "headerValue1" and
 queryStringParameters["QueryString1"]
            == "queryValue1" and stageVariables["StageVal1"] == "stageValue1"):
        response = generateAllow('me', event['methodArn'])
        print('authorized')
        return json.loads(response)
    else:
        print('unauthorized')
        raise Exception('Unauthorized') # Return a 401 Unauthorized response
        return 'unauthorized'

    # Help function to generate IAM policy


def generatePolicy(principalId, effect, resource):
    authResponse = {}
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument

    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }

    authResponse_JSON = json.dumps(authResponse)

    return authResponse_JSON


def generateAllow(principalId, resource):
    return generatePolicy(principalId, 'Allow', resource)


def generateDeny(principalId, resource):
    return generatePolicy(principalId, 'Deny', resource)
```

7. Choose **Deploy**.

After you create your Lambda function, you create and test a request-based Lambda authorizer in the API Gateway console.

To create a request-based Lambda authorizer

1. In the API Gateway console, create a simple API if you don't already have one.
2. Choose your API from the API list.
3. Choose **Authorizers**.
4. Choose **Create authorizer**.
5. For **Authorizer name**, enter a name.
6. For **Authorizer type**, select **Lambda**.

7. For **Lambda function**, select the AWS Region where you created your Lambda authorizer function, and then enter the function name.
8. Keep **Lambda invoke role** blank.
9. For **Lambda event payload**, select **Request**.
10. Under **Identity source type**, enter the following:

    a. Select **Header** and enter **`headerauth1`**, and then choose **Add parameter**.

    b. Under **Identity source type**, select **Query string** and enter **`QueryString1`**, and then choose **Add parameter**.

    c. Under **Identity source type**, select **Stage variable** and enter **`StageVar1`**.
11. Choose **Create authorizer**.

To test your authorizer

1. Select the name of the authorizer.
2. Under **Test authorizer**, enter the following:

    a. Select **Header** and enter **`headerValue1`**, and then choose **Add parameter**.

    b. Under **Identity source type**, select **Query string** and enter **`queryValue1`**, and then choose **Add parameter**.

    c. Under **Identity source type**, select **Stage variable** and enter **`stageValue1`**.
3. Choose **Test authorizer**.

In this example, the Lambda authorizer function checks the input parameters and acts as follows:

- If all the required parameter values match the expected values, the authorizer function returns a `200 OK` HTTP response and an IAM policy that looks like the following, and the method request succeeds:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Allow",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

- Otherwise, the authorizer function returns a `401 Unauthorized` HTTP response, and the method call fails.

> **Note**
>
> In production code, you may need to authenticate the user before granting authorization. If so, you can add authentication logic in the Lambda function as well by calling an authentication provider as directed in the documentation for that provider.

In addition to returning an IAM policy, the Lambda authorizer function must also return the caller's principal identifier. It can also optionally return a `context` object containing additional information that can be passed into the integration backend. For more information, see Output from an Amazon API Gateway Lambda authorizer.

Configure a Lambda authorizer using the API Gateway console

After you create the Lambda function and verify that it works, use the following steps to configure the API Gateway Lambda authorizer (formerly known as the custom authorizer) in the API Gateway console.

To configure a Lambda authorizer using the API Gateway console

1. Sign in to the API Gateway console.
2. Create a new or select an existing API, and then choose **Authorizers**.
3. Choose **Create authorizer**.
4. For **Authorizer name**, enter a name for the authorizer.
5. For **Authorizer type**, select **Lambda**.
6. For **Lambda function**, select the AWS Region where you created your Lambda authorizer function, and then enter the function name.
7. Keep **Lambda invoke role** blank to let the API Gateway console set a resource-based policy. The policy grants API Gateway permissions to invoke the authorizer Lambda function. You can

also choose to enter the name of an IAM role to allow API Gateway to invoke the authorizer Lambda function. For an example of such a role, see Create an assumable IAM role.

8. For **Lambda event payload**, select either **Token** for a TOKEN authorizer or **Request** for a REQUEST authorizer. (This is the same as setting the type property to TOKEN or REQUEST.)
9. Depending on the choice of the previous step, do one of the following:

   a. For the **Token** option, do the following:

      - For **Token source**, enter the header name that contains the authorization token. The API client must include a header of this name to send the authorization token to the Lambda authorizer.
      - Optionally, for **Token validation**, enter a RegEx statement. API Gateway performs initial validation of the input token against this expression and invokes the authorizer upon successful validation. This helps reduce calls to your API.
      - To cache the authorization policy generated by the authorizer, keep **Authorization caching** turned on. When policy caching is enabled, you can choose to modify the **TTL** value. Setting the **TTL** to zero disables policy caching. When policy caching is enabled, the header name specified in **Token source** becomes the cache key. If multiple values are passed to this header in the request, all values will become the cache key, with the order preserved.

        > **Note**
        >
        > The default **TTL** value is 300 seconds. The maximum value is 3600 seconds; this limit cannot be increased.

   b. For the **Request** option, do the following:

      - For **Identity source type**, select a parameter type. Supported parameter types are `Header`, `Query string`, `Stage variable`, and `Context`. To add more identity sources, choose **Add parameter**.
      - To cache the authorization policy generated by the authorizer, keep **Authorization caching** turned on. When policy caching is enabled, you can choose to modify the **TTL** value. Setting the **TTL** to zero disables policy caching.

        API Gateway uses the specified identity sources as the request authorizer caching key. When caching is enabled, API Gateway calls the authorizer's Lambda function only after successfully verifying that all the specified identity sources are present at runtime. If a specified identify source is missing, null, or empty, API Gateway returns a `401 Unauthorized` response without calling the authorizer Lambda function.

When multiple identity sources are defined, they are all used to derive the authorizer's cache key. Changing any of the cache key parts causes the authorizer to discard the cached policy document and generate a new one. If a header with multiple values is passed in the request, then all values will be part of the cache key, with the order preserved.

- When caching is turned off, it is not necessary to specify an identity source. API Gateway directly passes the request to the authorizer Lambda function.

> **Note**
>
> To enable caching, your authorizer must return a policy that is applicable to all methods across an API. To enforce method-specific policy, you can turn off **Authorization caching**.

10. Choose **Create authorizer**.
11. After the authorizer is created for the API, you can test the authorizer before it is configured on a method. To test an authorizer, select the name of the authorizer.
12. a. For the `TOKEN` authorizer, under **Token value**, enter a valid token. Choose **Test authorizer**. The token will be passed to the Lambda function as the header you specified in the **Token source** setting of the authorizer.

    b. For the `REQUEST` authorizer, under **Identity source type**, select a parameter type and enter a value. To add more parameters, select **Add parameter**. Choose **Test authorizer**.

In addition to using the API Gateway console, you can use AWS CLI or an AWS SDK for API Gateway to test invoking an authorizer. To do so using the AWS CLI, see test-invoke-authorizer.

> **Note**
>
> Test-invoke for method executions test-invoke for authorizers are independent processes.

To test invoking a method using the API Gateway console, see Use the console to test a REST API method. To test invoking a method using the AWS CLI, see test-invoke-method.

To test invoking a method and a configured authorizer, deploy the API, and then use cURL or Postman to call the method, providing the required token or request parameters.

The next procedure shows how to configure an API method to use the Lambda authorizer.

To configure an API method to use a Lambda authorizer

1. Choose **Resources**. Choose a new method or choose an existing method. If necessary, create a new resource.
2. On the **Method request** tab, under **Method request settings**, choose **Edit**.
3. For **Authorizer**, from the dropdown menu, select the Lambda authorizer you just created.
4. (Optional) If you want to pass the authorization token to the backend, choose **HTTP request headers**. Choose **Add header**, and then add the name of the authorization header. In **Name**, enter the header name that matches the **Token source** name you specified when you created the Lambda authorizer for the API. This step does not apply to `REQUEST` authorizers.
5. Choose **Save**.
6. Choose **Deploy API** to deploy the API to a stage. Note the **Invoke URL** value. You need it when calling the API. For a `REQUEST` authorizer using stage variables, you must also define the required stage variables and specify their values while on the **Stages** page.

Input to an Amazon API Gateway Lambda authorizer

`TOKEN` input format

For a Lambda authorizer (formerly known as a custom authorizer) of the `TOKEN` type, you must specify a custom header as the **Token Source** when you configure the authorizer for your API. The API client must pass the required authorization token in that header in the incoming request. Upon receiving the incoming method request, API Gateway extracts the token from the custom header. It then passes the token as the `authorizationToken` property of the `event` object of the Lambda function, in addition to the method ARN as the `methodArn` property:

```
{
    "type":"TOKEN",
    "authorizationToken":"{caller-supplied-token}",
    "methodArn":"arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/
[{resource}/[{child-resources}]]"
}
```

In this example, the `type` property specifies the authorizer type, which is a `TOKEN` authorizer. The *`{caller-supplied-token}`* originates from the authorization header in a client request, and can be any string value. The `methodArn` is the ARN of the incoming method request and is populated by API Gateway in accordance with the Lambda authorizer configuration.

REQUEST input format

For a Lambda authorizer of the `REQUEST` type, API Gateway passes request parameters to the authorizer Lambda function as part of the `event` object. The request parameters include headers, path parameters, query string parameters, stage variables, and some of request context variables. The API caller can set the path parameters, headers, and query string parameters. The API developer must set the stage variables during the API deployment and API Gateway provides the request context at run time.

> **Note**
>
> Path parameters can be passed as request parameters to the Lambda authorizer function, but they cannot be used as identity sources.

The following example shows an input to a `REQUEST` authorizer for an API method (`GET /request`) with a proxy integration:

```
{
  "type": "REQUEST",
  "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/request",
  "resource": "/request",
  "path": "/request",
  "httpMethod": "GET",
  "headers": {
    "X-AMZ-Date": "20170718T062915Z",
    "Accept": "*/*",
    "HeaderAuth1": "headerValue1",
    "CloudFront-Viewer-Country": "US",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Is-Mobile-Viewer": "false",
    "User-Agent": "..."
  },
  "queryStringParameters": {
    "QueryString1": "queryValue1"
  },
  "pathParameters": {},
  "stageVariables": {
    "StageVar1": "stageValue1"
  },
  "requestContext": {
    "path": "/request",
    "accountId": "123456789012",
    "resourceId": "05c7jb",
    "stage": "test",
    "requestId": "...",
    "identity": {
      "apiKey": "...",
      "sourceIp": "...",
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "resourcePath": "/request",
    "httpMethod": "GET",
    "apiId": "abcdef123"
  }
}
```

The `requestContext` is a map of key-value pairs and corresponds to the $context variable. Its outcome is API-dependent. API Gateway may add new keys to the map. For more information about Lambda function input in Lambda proxy integration, see Input format of a Lambda function for proxy integration.

Output from an Amazon API Gateway Lambda authorizer

A Lambda authorizer function's output is a dictionary-like object, which must include the principal identifier (`principalId`) and a policy document (`policyDocument`) containing a list of policy statements. The output can also include a `context` map containing key-value pairs. If the API uses a usage plan (the [apiKeySource](#) is set to `AUTHORIZER`), the Lambda authorizer function must return one of the usage plan's API keys as the `usageIdentifierKey` property value.

The following shows an example of this output.

```
{
  "principalId": "yyyyyyyy", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-
api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "stringKey": "value",
    "numberKey": "1",
    "booleanKey": "true"
  },
  "usageIdentifierKey": "{api-key}"
}
```

Here, a policy statement specifies whether to allow or deny (`Effect`) the API Gateway execution service to invoke (`Action`) the specified API method (`Resource`). You can use a wild card (*) to specify a resource type (method). For information about setting valid policies for calling an API, see [Statement reference of IAM policies for executing API in API Gateway](#).

For an authorization-enabled method ARN, e.g., `arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/[{resource}/[{child-resources}]]`, the maximum length is 1600 bytes. The path parameter values, the size of which are determined at run time, can cause the ARN length to exceed the limit. When this happens, the API client will receive a `414 Request URI too long` response.

In addition, the Resource ARN, as shown in the policy statement output by the authorizer, is currently limited to 512 characters long. For this reason, you must not use URI with a JWT token of a significant length in a request URI. You can safely pass the JWT token in a request header, instead.

You can access the `principalId` value in a mapping template using the `$context.authorizer.principalId` variable. This is useful if you want to pass the value to the backend. For more information, see `$context` Variables for data models, authorizers, mapping templates, and CloudWatch access logging.

You can access the `stringKey`, `numberKey`, or `booleanKey` value (for example, `"value"`, `"1"`, or `"true"`) of the `context` map in a mapping template by calling `$context.authorizer.stringKey`, `$context.authorizer.numberKey`, or `$context.authorizer.booleanKey`, respectively. The returned values are all stringified. Notice that you cannot set a JSON object or array as a valid value of any key in the `context` map.

You can use the `context` map to return cached credentials from the authorizer to the backend, using an integration request mapping template. This enables the backend to provide an improved user experience by using the cached credentials to reduce the need to access the secret keys and open the authorization tokens for every request.

For the Lambda proxy integration, API Gateway passes the `context` object from a Lambda authorizer directly to the backend Lambda function as part of the input `event`. You can retrieve the `context` key-value pairs in the Lambda function by calling `$event.requestContext.authorizer.`*`key`*.

`{api-key}` stands for an API key in the API stage's usage plan. For more information, see the section called “Usage plans”.

The following shows example output from the example Lambda authorizer. The example output contains a policy statement to block (`Deny`) calls to the `GET` method for the `dev` stage of an API (`ymy8tbxw7b`) of an AWS account (`123456789012`).

```
{
  "principalId": "user",
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Deny",
        "Resource": "arn:aws:execute-api:us-west-2:123456789012:ymy8tbxw7b/dev/GET/"
      }
    ]
  }
}
```

Call an API with API Gateway Lambda authorizers

Having configured the Lambda authorizer (formerly known as the custom authorizer) and deployed the API, you should test the API with the Lambda authorizer enabled. For this, you need a REST client, such as cURL or Postman. For the following examples, we use Postman.

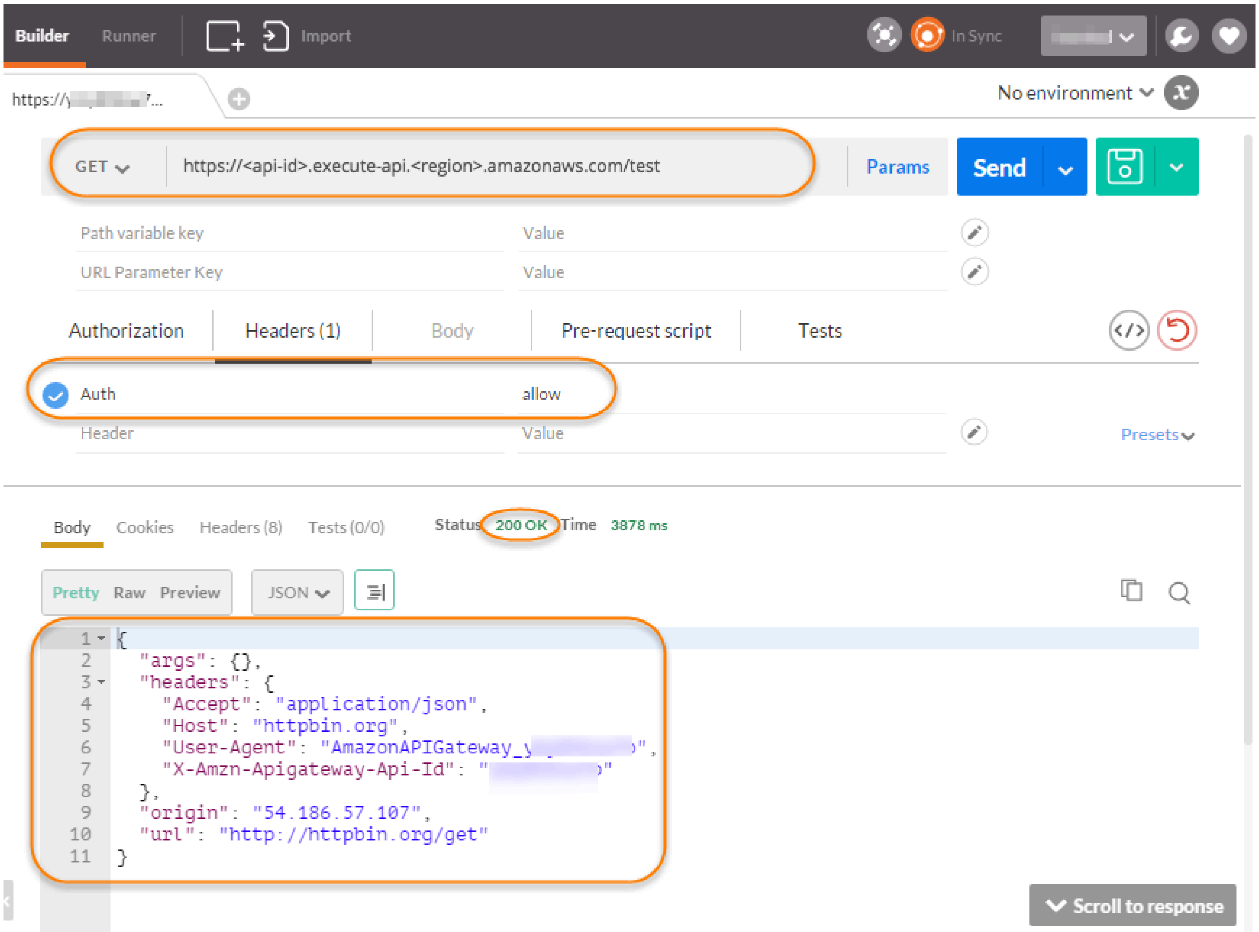
> **Note**
>
> When calling an authorizer-enabled method, API Gateway does not log the call to CloudWatch if the required token for the `TOKEN` authorizer is not set, is null, or is invalidated by the specified **Token validation expression**. Similarly, API Gateway does not log the call to CloudWatch if any of the required identity sources for the `REQUEST` authorizer are not set, are null, or are empty.

In the following, we show how to use Postman to call or test an API with a Lambda `TOKEN` authorizer. The method can be applied to calling an API with a Lambda `REQUEST` authorizer, if you specify the required path, header, or query string parameters explicitly.

To call an API with the custom `TOKEN` authorizer

1. Open **Postman**, choose the **GET** method, and paste the API's **Invoke URL** into the adjacent URL field.

   Add the Lambda authorization token header and set the value to `allow`. Choose **Send**.



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

Path variable key	Value
URL Parameter Key	Value

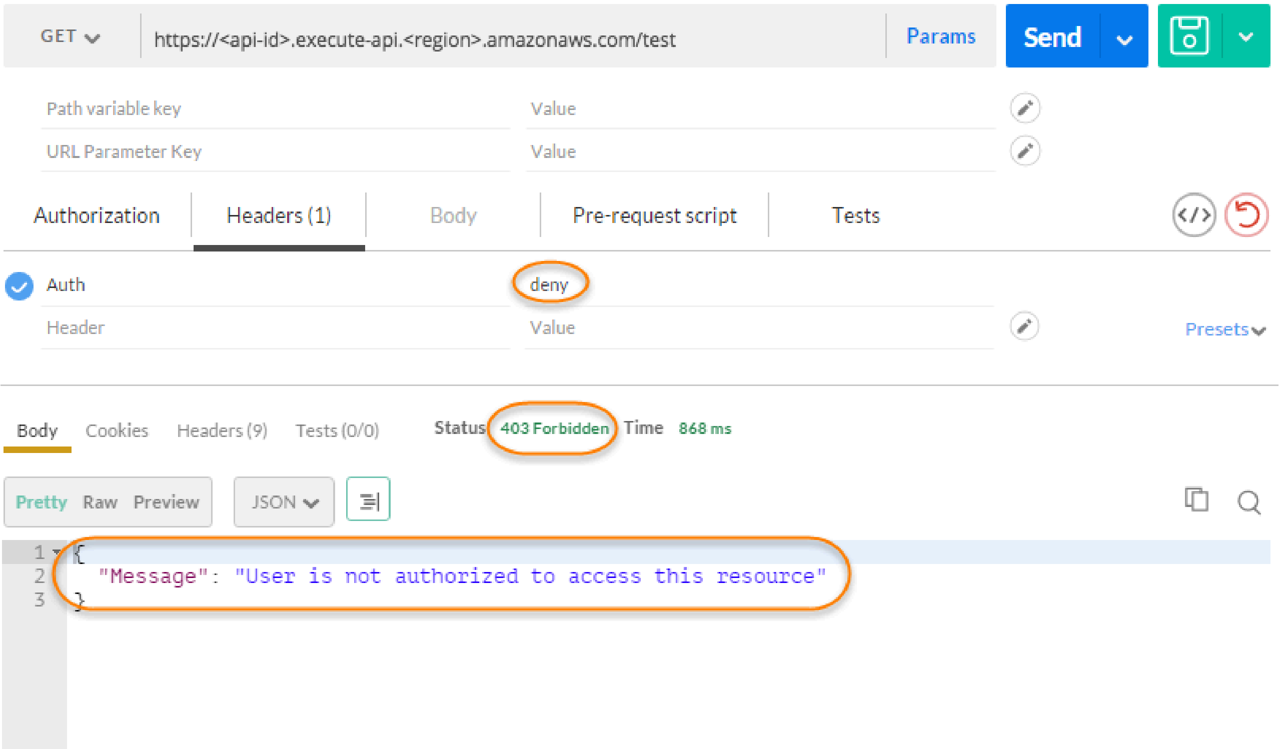
Header	Value
☑ Auth	allow
Header	Value

Status 200 OK Time 3878 ms

```
{
  "args": {},
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "AmazonAPIGateway_[redacted]",
    "X-Amzn-Apigateway-Api-Id": "[redacted]"
  },
  "origin": "54.186.57.107",
  "url": "http://httpbin.org/get"
}
```

The response shows that the API Gateway Lambda authorizer returns a **200 OK** response and successfully authorizes the call to access the HTTP endpoint (http://httpbin.org/get) integrated with the method.

2. Still in Postman, change the Lambda authorization token header value to `deny`. Choose **Send**.



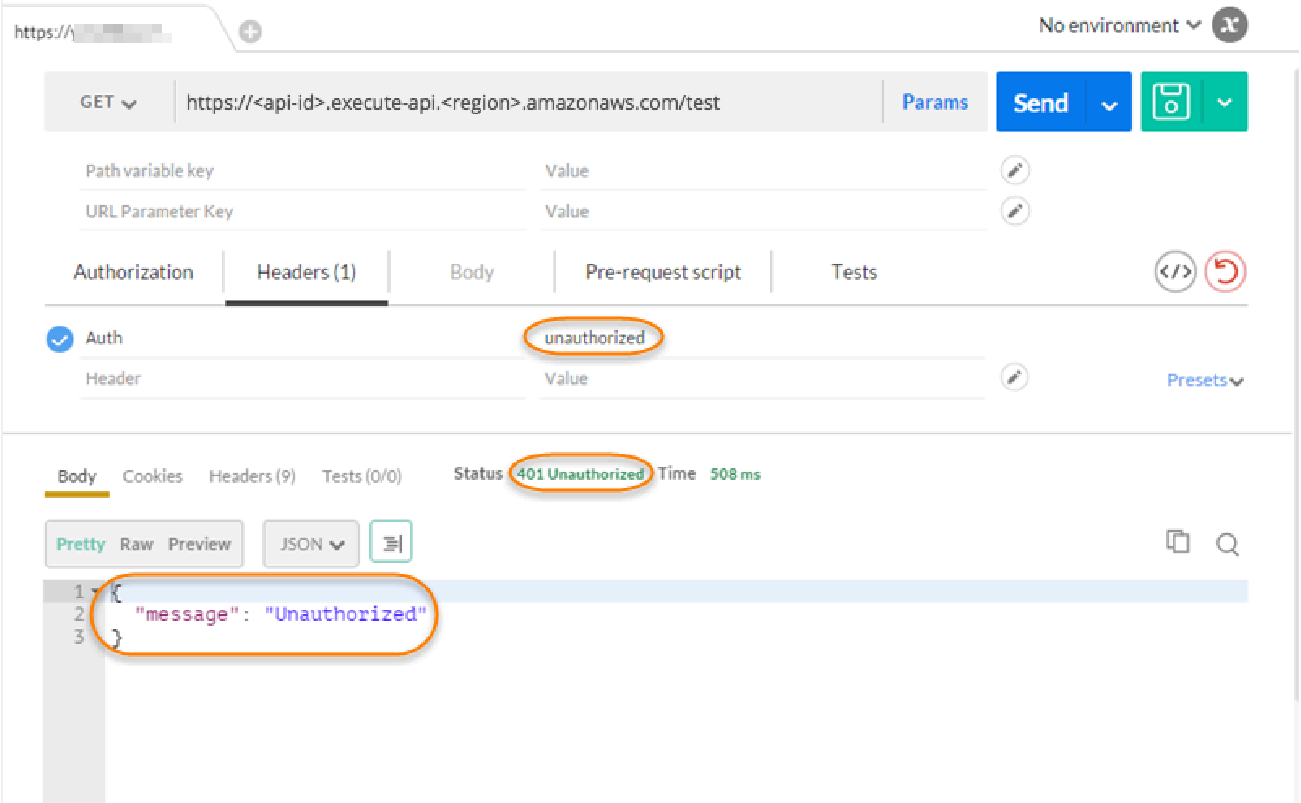
Auth: deny

Status 403 Forbidden Time 868 ms

```
{
  "Message": "User is not authorized to access this resource"
}
```

The response shows that the API Gateway Lambda authorizer returns a **403 Forbidden** response without authorizing the call to access the HTTP endpoint.

3. In Postman, change the Lambda authorization token header value to `unauthorized` and choose **Send**.



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

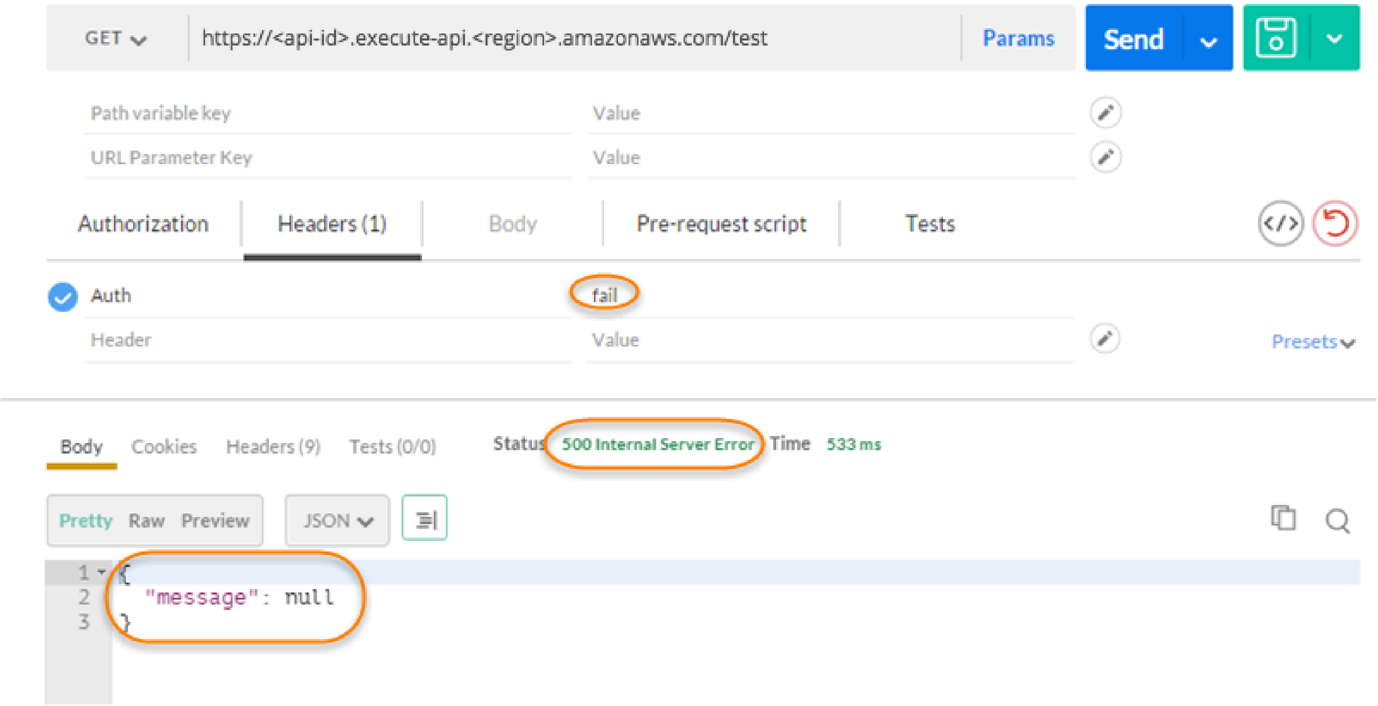
Auth: unauthorized

Status 401 Unauthorized Time 508 ms

```
{
  "message": "Unauthorized"
}
```

The response shows that API Gateway returns a **401 Unauthorized** response without authorizing the call to access the HTTP endpoint.

4. Now, change the Lambda authorization token header value to `fail`. Choose **Send**.



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

Auth: fail

Status 500 Internal Server Error Time 533 ms

```
{
  "message": null
}
```

The response shows that API Gateway returns a **500 Internal Server Error** response without authorizing the call to access the HTTP endpoint.

Configure a cross-account Lambda authorizer

You can now also use an AWS Lambda function from a different AWS account as your API authorizer function. Each account can be in any region where Amazon API Gateway is available. The Lambda authorizer function can use bearer token authentication strategies such as OAuth or SAML. This makes it easy to centrally manage and share a central Lambda authorizer function across multiple API Gateway APIs.

In this section, we show how to configure a cross-account Lambda authorizer function using the Amazon API Gateway console.

These instructions assume that you already have an API Gateway API in one AWS account and a Lambda authorizer function in another account.

Configure a cross-account Lambda authorizer using the API Gateway console

Log in to the Amazon API Gateway console in the account that has your API in it, and then do the following:

1. Choose your API, and then in the main navigation pane, choose **Authorizers**.
2. Choose **Create authorizer**.
3. For **Authorizer name**, enter a name for the authorizer.
4. For **Authorizer type**, select **Lambda**.
5. For **Lambda Function**, enter the full ARN for the Lambda authorizer function that you have in your second account.

   > **Note**
   >
   > In the Lambda console, you can find the ARN for your function in the upper right corner of the console window.

6. A warning with an `aws lambda add-permission` command string will appear. This policy grants API Gateway permission to invoke the authorizer Lambda function. Copy the command and save it for later. You run the command after you create the authorizer.

7. Keep **Lambda invoke role** blank to let the API Gateway console set a resource-based policy. The policy grants API Gateway permission to invoke the authorizer Lambda function. You can also choose to enter an IAM role to allow API Gateway to invoke the authorizer Lambda function. For an example of such a role, see Create an assumable IAM role.

8. For **Lambda event payload**, select either **Token** for a `TOKEN` authorizer or **Request** for a `REQUEST` authorizer.

9. Depending on the choice of the previous step, do one of the following:

   a. For the **Token** option, do the following:

      - For **Token source**, enter the header name that contains the authorization token. The API client must include a header of this name to send the authorization token to the Lambda authorizer.
      - Optionally, for **Token validation**, enter a RegEx statement. API Gateway performs initial validation of the input token against this expression and invokes the authorizer upon successful validation. This helps reduce calls to your API.
      - To cache the authorization policy generated by the authorizer, keep **Authorization caching** turned on. When policy caching is enabled, you can choose to modify the **TTL** value. Setting the **TTL** to zero disables policy caching. When policy caching is enabled, the header name specified in **Token source** becomes the cache key. If multiple values are passed to this header in the request, all values will become the cache key, with the order preserved.

      > **Note**
      >
      > The default **TTL** value is 300 seconds. The maximum value is 3600 seconds; this limit cannot be increased.

   b. For the **Request** option, do the following:

      - For **Identity source type**, select a parameter type. Supported parameter types are `Header`, `Query string`, `Stage variable`, and `Context`. To add more identity sources, choose **Add parameter**.
      - To cache the authorization policy generated by the authorizer, keep **Authorization caching** turned on. When policy caching is enabled, you can choose to modify the **TTL** value. Setting the **TTL** to zero disables policy caching.

API Gateway uses the specified identity sources as the request authorizer caching key. When caching is enabled, API Gateway calls the authorizer's Lambda function only after successfully verifying that all the specified identity sources are present at runtime. If a specified identify source is missing, null, or empty, API Gateway returns a `401 Unauthorized` response without calling the authorizer Lambda function.

When multiple identity sources are defined, they are all used to derive the authorizer's cache key. Changing any of the cache key parts causes the authorizer to discard the cached policy document and generate a new one. If a header with multiple values is passed in the request, then all values will be part of the cache key, with the order preserved.

- When caching is turned off, it is not necessary to specify an identity source.

> **Note**
>
> To enable caching, your authorizer must return a policy that is applicable to all methods across an API. To enforce method-specific policy, you can turn off **Authorization caching**.

10. Choose **Create authorizer**.
11. Paste the `aws lambda add-permission` command string that you copied in a previous step into an AWS CLI window that is configured for your second account. Replace `AUTHORIZER_ID` with your authorizer's ID. This will grant your first account access to your second account's Lambda authorizer function.

Control access to a REST API using Amazon Cognito user pools as authorizer

As an alternative to using IAM roles and policies or Lambda authorizers (formerly known as custom authorizers), you can use an Amazon Cognito user pool to control who can access your API in Amazon API Gateway.

To use an Amazon Cognito user pool with your API, you must first create an authorizer of the `COGNITO_USER_POOLS` type and then configure an API method to use that authorizer. After the API is deployed, the client must first sign the user in to the user pool, obtain an identity or access token for the user, and then call the API method with one of the tokens, which are typically set to the request's `Authorization` header. The API call succeeds only if the required token is supplied and the supplied token is valid, otherwise, the client isn't authorized to make the call because the client did not have credentials that could be authorized.

The identity token is used to authorize API calls based on identity claims of the signed-in user. The access token is used to authorize API calls based on the custom scopes of specified access-protected resources. For more information, see Using Tokens with User Pools and Resource Server and Custom Scopes.

To create and configure an Amazon Cognito user pool for your API, you perform the following tasks:

- Use the Amazon Cognito console, CLI/SDK, or API to create a user pool—or use one that's owned by another AWS account.
- Use the API Gateway console, CLI/SDK, or API to create an API Gateway authorizer with the chosen user pool.
- Use the API Gateway console, CLI/SDK, or API to enable the authorizer on selected API methods.

To call any API methods with a user pool enabled, your API clients perform the following tasks:

- Use the Amazon Cognito CLI/SDK or API to sign a user in to the chosen user pool, and obtain an identity token or access token. To learn more about using the SDKs, see Code examples for Amazon Cognito using AWS SDKs.
- Use a client-specific framework to call the deployed API Gateway API and supply the appropriate token in the `Authorization` header.

As the API developer, you must provide your client developers with the user pool ID, a client ID, and possibly the associated client secrets that are defined as part of the user pool.

Note

To let a user sign in using Amazon Cognito credentials and also obtain temporary credentials to use with the permissions of an IAM role, use Amazon Cognito Federated Identities. For each API resource endpoint HTTP method, set the authorization type, category `Method Execution`, to `AWS_IAM`.

In this section, we describe how to create a user pool, how to integrate an API Gateway API with the user pool, and how to invoke an API that's integrated with the user pool.

Topics

- Obtain permissions to create Amazon Cognito user pool authorizers for a REST API
- Create an Amazon Cognito user pool for a REST API
- Integrate a REST API with an Amazon Cognito user pool
- Call a REST API integrated with an Amazon Cognito user pool
- Configure cross-account Amazon Cognito authorizer for a REST API using the API Gateway console
- Create an Amazon Cognito authorizer for a REST API using AWS CloudFormation

Obtain permissions to create Amazon Cognito user pool authorizers for a REST API

To create an authorizer with an Amazon Cognito user pool, you must have `Allow` permissions to create or update an authorizer with the chosen Amazon Cognito user pool. The following IAM policy document shows an example of such permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST"
            ],
            "Resource": "arn:aws:apigateway:*::/restapis/*/authorizers",
            "Condition": {
                "ArnLike": {
                    "apigateway:CognitoUserPoolProviderArn": [
                        "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-
east-1_aD06NQmjO",
                        "arn:aws:cognito-idp:us-east-1:234567890123:userpool/us-
east-1_xJ1MQtPEN"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:PATCH"
            ],
            "Resource": "arn:aws:apigateway:*::/restapis/*/authorizers/*",
            "Condition": {
                "ArnLike": {
                    "apigateway:CognitoUserPoolProviderArn": [
                        "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-
east-1_aD06NQmjO",
                        "arn:aws:cognito-idp:us-east-1:234567890123:userpool/us-
east-1_xJ1MQtPEN"
                    ]
                }
            }
        }
    ]
}
```

Make sure that the policy is attached to an IAM group that you belong to or an IAM role that you're assigned to.

In the preceding policy document, the `apigateway:POST` action is for creating a new authorizer, and the `apigateway:PATCH` action is for updating an existing authorizer. You can restrict the policy to a specific region or a particular API by overriding the first two wildcard (*) characters of the `Resource` values, respectively.

The `Condition` clauses that are used here are to restrict the `Allowed` permissions to the specified user pools. When a `Condition` clause is present, access to any user pools that don't match the conditions is denied. When a permission doesn't have a `Condition` clause, access to any user pool is allowed.

You have the following options to set the `Condition` clause:

- You can set an `ArnLike` or `ArnEquals` conditional expression to permit creating or updating `COGNITO_USER_POOLS` authorizers with the specified user pools only.
- You can set an `ArnNotLike` or `ArnNotEquals` conditional expression to permit creating or updating `COGNITO_USER_POOLS` authorizers with any user pool that isn't specified in the expression.
- You can omit the `Condition` clause to permit creating or updating `COGNITO_USER_POOLS` authorizers with any user pool, of any AWS account, and in any region.

For more information on the Amazon Resource Name (ARN) conditional expressions, see Amazon Resource Name Condition Operators. As shown in the example, `apigateway:CognitoUserPoolProviderArn` is a list of ARNs of the `COGNITO_USER_POOLS` user pools that can or can't be used with an API Gateway authorizer of the `COGNITO_USER_POOLS` type.

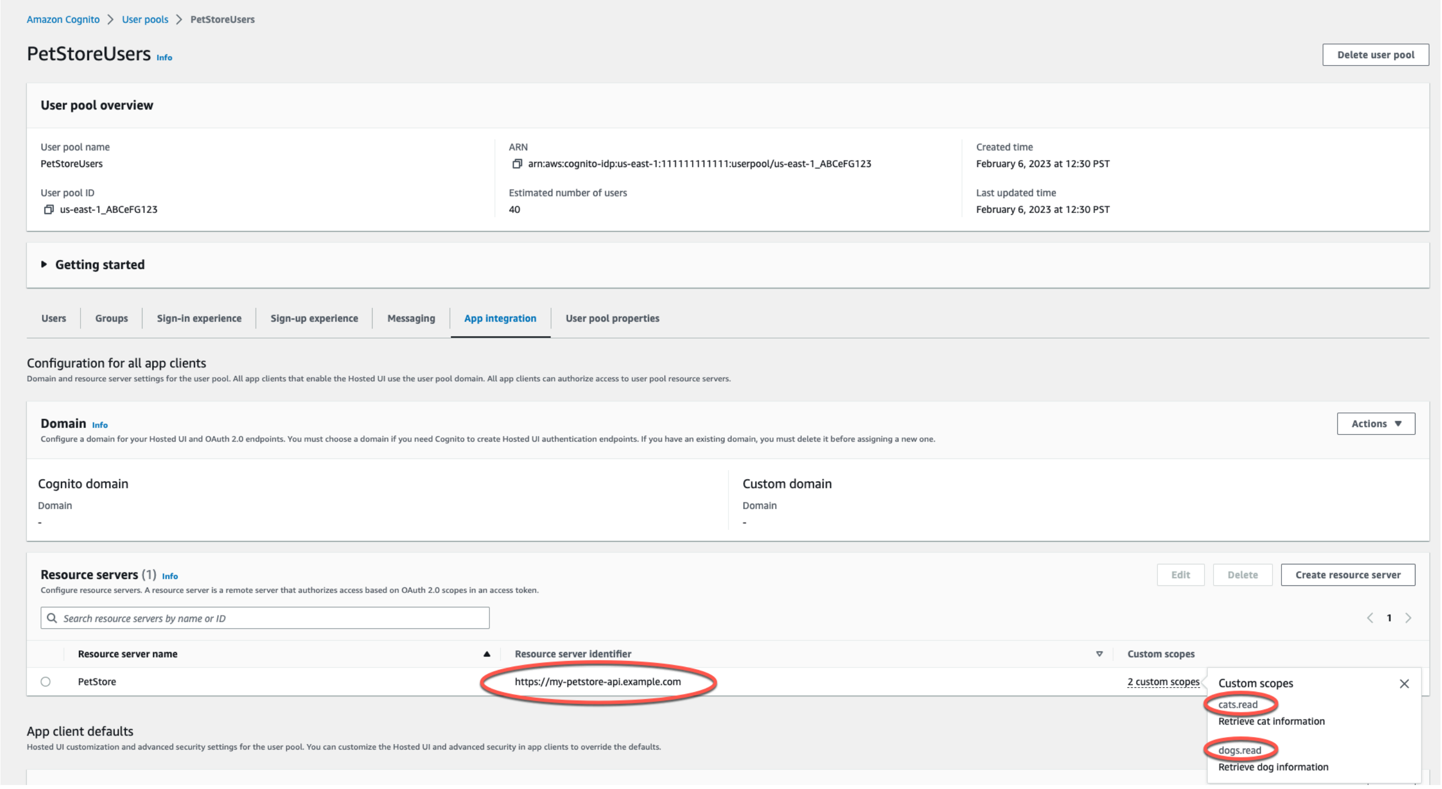
Create an Amazon Cognito user pool for a REST API

Before integrating your API with a user pool, you must create the user pool in Amazon Cognito. Your user pool configuration must follow all resource quotas for Amazon Cognito. All user-defined Amazon Cognito variables such as groups, users, and roles should use only alphanumeric characters. For instructions on how to create a user pool, see Tutorial: Creating a user pool in the *Amazon Cognito Developer Guide*.

Note the user pool ID, client ID, and any client secret. The client must provide them to Amazon Cognito for the user to register with the user pool, to sign in to the user pool, and to obtain an identity or access token to be included in requests to call API methods that are configured with the user pool. Also, you must specify the user pool name when you configure the user pool as an authorizer in API Gateway, as described next.

If you're using access tokens to authorize API method calls, be sure to configure the app integration with the user pool to set up the custom scopes that you want on a given resource server. For more information about using tokens with Amazon Cognito user pools, see Using Tokens with User Pools. For more information about resource servers, see Defining Resource Servers for Your User Pool.

Note the configured resource server identifiers and custom scope names. You need them to construct the access scope full names for **OAuth Scopes**, which is used by the `COGNITO_USER_POOLS` authorizer.



PetStoreUsers

User pool overview

User pool name
PetStoreUsers

User pool ID
us-east-1_ABCeFG123

ARN
arn:aws:cognito-idp:us-east-1:111111111111:userpool/us-east-1_ABCeFG123

Estimated number of users
40

Created time
February 6, 2023 at 12:30 PST

Last updated time
February 6, 2023 at 12:30 PST

Configuration for all app clients

Domain and resource server settings for the user pool. All app clients that enable the Hosted UI use the user pool domain. All app clients can authorize access to user pool resource servers.

Domain

Configure a domain for your Hosted UI and OAuth 2.0 endpoints. You must choose a domain if you need Cognito to create Hosted UI authentication endpoints. If you have an existing domain, you must delete it before assigning a new one.

Cognito domain
Domain
-

Custom domain
Domain
-

Resource servers (1)

Configure resource servers. A resource server is a remote server that authorizes access based on OAuth 2.0 scopes in an access token.

	Resource server name	Resource server identifier	Custom scopes
○	PetStore	https://my-petstore-api.example.com	2 custom scopes

Custom scopes
cats.read
Retrieve cat information
dogs.read
Retrieve dog information

App client defaults

Hosted UI customization and advanced security settings for the user pool. You can customize the Hosted UI and advanced security in app clients to override the defaults.

Integrate a REST API with an Amazon Cognito user pool

After creating an Amazon Cognito user pool, in API Gateway, you must then create a COGNITO_USER_POOLS authorizer that uses the user pool. The following procedure shows you how to do this using the API Gateway console.

Note

You can use the CreateAuthorizer action to create a COGNITO_USER_POOLS authorizer that uses multiple user pools. You can use up to 1,000 user pools for one COGNITO_USER_POOLS authorizer. This limit cannot be increased.

Important

After performing any of the procedures below, you'll need to deploy or redeploy your API to propagate the changes. For more information about deploying your API, see Deploying a REST API in Amazon API Gateway.

To create a COGNITO_USER_POOLS authorizer by using the API Gateway console

1. Create a new API, or select an existing API in API Gateway.
2. In the main navigation pane, choose **Authorizers**.
3. Choose **Create authorizer**.
4. To configure the new authorizer to use a user pool, do the following:

    a. For **Authorizer name**, enter a name.

    b. For **Authorizer type**, select **Cognito**.

    c. For **Cognito user pool**, choose the AWS Region where you created your Amazon Cognito and select an available user pool.

    d. For **Token source**, enter **`Authorization`** as the header name to pass the identity or access token that's returned by Amazon Cognito when a user signs in successfully.

    e. (Optional) Enter a regular expression in the **Token validation** field to validate the `aud` (audience) field of the identity token before the request is authorized with Amazon Cognito. Note that when using an access token this validation rejects the request due to the access token not containing the `aud` field.

    f. Choose **Create authorizer**.
5. After creating the `COGNITO_USER_POOLS` authorizer, you can optionally test invoke it by supplying an identity token that's provisioned from the user pool. You can obtain this identity token by calling the Amazon Cognito Identity SDK to perform user sign-in. You can also use the `InitiateAuth` action. If you do not configure any **Authorization scopes**, API Gateway treats the supplied token as an identity token.

The preceding procedure creates a `COGNITO_USER_POOLS` authorizer that uses the newly created Amazon Cognito user pool. Depending on how you enable the authorizer on an API method, you can use either an identity token or an access token that's provisioned from the integrated user pool.

To configure a COGNITO_USER_POOLS authorizer on methods

1. Choose **Resources**. Choose a new method or choose an existing method. If necessary, create a resource.
2. On the **Method request** tab, under **Method request settings**, choose **Edit**.

3. For **Authorizer**, from the dropdown menu, select the **Amazon Cognito user pool authorizers** you just created.

4. To use an identity token, do the following:

   a. Keep **Authorization Scopes** empty.

   b. If needed, in the **Integration request**, add the `$context.authorizer.claims['`*`property-name`*`']` or `$context.authorizer.claims.`*`property-name`* expressions in a body-mapping template to pass the specified identity claims property from the user pool to the backend. For simple property names, such as `sub` or `custom-sub`, the two notations are identical. For complex property names, such as `custom:role`, you can't use the dot notation. For example, the following mapping expressions pass the claim's standard fields of `sub` and `email` to the backend:

```
{
 "context" : {
  "sub" : "$context.authorizer.claims.sub",
  "email" : "$context.authorizer.claims.email"
 }
}
```

   If you declared a custom claim field when you configured a user pool, you can follow the same pattern to access the custom fields. The following example gets a custom `role` field of a claim:

```
{
 "context" : {
  "role" : "$context.authorizer.claims.role"
    }
}
```

   If the custom claim field is declared as `custom:role`, use the following example to get the claim's property:

```
{
 "context" : {
  "role" : "$context.authorizer.claims['custom:role']"
    }
}
```

5. To use an access token, do the following:

   a. For **Authorization Scopes**, enter one or more full names of a scope that has been configured when the Amazon Cognito user pool was created. For example, following the example given in Create an Amazon Cognito user pool for a REST API, one of the scopes is `https://my-petstore-api.example.com/cats.read`.

      At runtime, the method call succeeds if any scope that's specified on the method in this step matches a scope that's claimed in the incoming token. Otherwise, the call fails with a `401 Unauthorized` response.

   b. Choose **Save**.

6. Repeat these steps for other methods that you choose.

With the `COGNITO_USER_POOLS` authorizer, if the **OAuth Scopes** option isn't specified, API Gateway treats the supplied token as an identity token and verifies the claimed identity against the one from the user pool. Otherwise, API Gateway treats the supplied token as an access token and verifies the access scopes that are claimed in the token against the authorization scopes declared on the method.

Instead of using the API Gateway console, you can also enable an Amazon Cognito user pool on a method by specifying an OpenAPI definition file and importing the API definition into API Gateway.

To import a COGNITO_USER_POOLS authorizer with an OpenAPI definition file

1. Create (or export) an OpenAPI definition file for your API.
2. Specify the `COGNITO_USER_POOLS` authorizer (`MyUserPool`) JSON definition as part of the `securitySchemes` section in OpenAPI 3.0 or the `securityDefinitions` section in Open API 2.0 as follows:

   OpenAPI 3.0

   ```
     "securitySchemes": {
       "MyUserPool": {
         "type": "apiKey",
         "name": "Authorization",
         "in": "header",
         "x-amazon-apigateway-authtype": "cognito_user_pools",
   ```

```
      "x-amazon-apigateway-authorizer": {
        "type": "cognito_user_pools",
        "providerARNs": [
          "arn:aws:cognito-idp:{region}:{account_id}:userpool/{user_pool_id}"
        ]
      }
    }
```

OpenAPI 2.0

```
  "securityDefinitions": {
    "MyUserPool": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "cognito_user_pools",
      "x-amazon-apigateway-authorizer": {
        "type": "cognito_user_pools",
        "providerARNs": [
          "arn:aws:cognito-idp:{region}:{account_id}:userpool/{user_pool_id}"
        ]
      }
    }
```

3. To use the identity token for method authorization, add `{ "MyUserPool": [] }` to the `security` definition of the method, as shown in the following GET method on the root resource.

```
  "paths": {
    "/": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "text/html"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "security": [
          {
            "MyUserPool": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "type": "mock",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type": "'text/html'"
              },
            }
          },
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match"
        }
      },
      ...
   }
```

4. To use the access token for method authorization, change the above security definition to `{ "MyUserPool": [resource-server/scope, ...] }`:

```
    "paths": {
      "/": {
        "get": {
          "consumes": [
            "application/json"
          ],
          "produces": [
            "text/html"
          ],
          "responses": {
            "200": {
              "description": "200 response",
              "headers": {
                "Content-Type": {
                  "type": "string"
                }
              }
            }
          },
          "security": [
            {
              "MyUserPool": ["https://my-petstore-api.example.com/cats.read",
   "http://my.resource.com/file.read"]
            }
          ],
          "x-amazon-apigateway-integration": {
            "type": "mock",
            "responses": {
              "default": {
                "statusCode": "200",
                "responseParameters": {
                  "method.response.header.Content-Type": "'text/html'"
                },
              }
            },
            "requestTemplates": {
              "application/json": "{\"statusCode\": 200}"
            },
            "passthroughBehavior": "when_no_match"
          }
        },
        ...
     }
```

5. If needed, you can set other API configuration settings by using the appropriate OpenAPI definitions or extensions. For more information, see Working with API Gateway extensions to OpenAPI.

Call a REST API integrated with an Amazon Cognito user pool

To call a method with a user pool authorizer configured, the client must do the following:

- Enable the user to sign up with the user pool.
- Enable the user to sign in to the user pool.

- Obtain an identity or access token of the signed-in user from the user pool.
- Include the token in the `Authorization` header (or another header you specified when you created the authorizer).

You can use AWS Amplify to perform these tasks. See Integrating Amazon Cognito With Web and Mobile Apps for more information.

- For Android, see Getting Started with Amplify for Android.
- To use iOS see Getting started with Amplify for iOS.
- To use JavaScript, see Getting Started with Amplify for Javascript.

Configure cross-account Amazon Cognito authorizer for a REST API using the API Gateway console

You can now also use a Amazon Cognito user pool from a different AWS account as your API authorizer. Each account can be in any region where Amazon API Gateway is available. The Amazon Cognito user pool can use bearer token authentication strategies such as OAuth or SAML. This makes it easy to centrally manage and share a central Amazon Cognito user pool authorizer across multiple API Gateway APIs.

In this section, we show how to configure a cross-account Amazon Cognito user pool using the Amazon API Gateway console.

These instructions assume that you already have an API Gateway API in one AWS account and a Amazon Cognito user pool in another account.

Configure cross-account Amazon Cognito authorizer using the API Gateway console

Log in to the Amazon API Gateway console in the account that has your API in it, and then do the following:

1. Create a new API, or select an existing API in API Gateway.
2. In the main navigation pane, choose **Authorizers**.
3. Choose **Create authorizer**.
4. To configure the new authorizer to use a user pool, do the following:

    a. For **Authorizer name**, enter a name.

    b. For **Authorizer type**, select **Cognito**.

c. For **Cognito user pool**, enter the full ARN for the user pool that you have in your second account.

> **Note**
>
> In the Amazon Cognito console, you can find the ARN for your user pool in the **Pool ARN** field of the **General Settings** pane.

d. For **Token source**, enter **`Authorization`** as the header name to pass the identity or access token that's returned by Amazon Cognito when a user signs in successfully.

e. (Optional) Enter a regular expression in the **Token validation** field to validate the aud (audience) field of the identity token before the request is authorized with Amazon Cognito. Note that when using an access token this validation rejects the request due to the access token not containing the aud field.

f. Choose **Create authorizer**.

Create an Amazon Cognito authorizer for a REST API using AWS CloudFormation

You can use AWS CloudFormation to create an Amazon Cognito user pool and an Amazon Cognito authorizer. The example AWS CloudFormation template does the following:

- Create an Amazon Cognito user pool. The client must first sign the user in to the user pool and obtain an identity or access token. If you're using access tokens to authorize API method calls, be sure to configure the app integration with the user pool to set up the custom scopes that you want on a given resource server.
- Creates an API Gateway API with a GET method.
- Creates an Amazon Cognito authorizer that uses the `Authorization` header as the token source.

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  UserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      AccountRecoverySetting:
        RecoveryMechanisms:
          - Name: verified_phone_number
```

```
            Priority: 1
          - Name: verified_email
            Priority: 2
      AdminCreateUserConfig:
        AllowAdminCreateUserOnly: true
      EmailVerificationMessage: The verification code to your new account is {####}
      EmailVerificationSubject: Verify your new account
      SmsVerificationMessage: The verification code to your new account is {####}
      VerificationMessageTemplate:
        DefaultEmailOption: CONFIRM_WITH_CODE
        EmailMessage: The verification code to your new account is {####}
        EmailSubject: Verify your new account
        SmsMessage: The verification code to your new account is {####}
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
  CogAuthorizer:
    Type: AWS::ApiGateway::Authorizer
    Properties:
      Name: CognitoAuthorizer
      RestApiId:
        Ref: Api
      Type: COGNITO_USER_POOLS
      IdentitySource: method.request.header.Authorization
      ProviderARNs:
        - Fn::GetAtt:
            - UserPool
            - Arn
  Api:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: MyCogAuthApi
  ApiDeployment:
    Type: AWS::ApiGateway::Deployment
    Properties:
      RestApiId:
        Ref: Api
    DependsOn:
      - CogAuthorizer
      - ApiGET
  ApiDeploymentStageprod:
    Type: AWS::ApiGateway::Stage
    Properties:
      RestApiId:
        Ref: Api
      DeploymentId:
        Ref: ApiDeployment
      StageName: prod
  ApiGET:
    Type: AWS::ApiGateway::Method
    Properties:
      HttpMethod: GET
      ResourceId:
        Fn::GetAtt:
          - Api
          - RootResourceId
      RestApiId:
        Ref: Api
      AuthorizationType: COGNITO_USER_POOLS
      AuthorizerId:
        Ref: CogAuthorizer
      Integration:
        IntegrationHttpMethod: GET
        Type: HTTP_PROXY
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets
Outputs:
  ApiEndpoint:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: Api
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
          - Ref: ApiDeploymentStageprod
          - /
```

Setting up REST API integrations

After setting up an API method, you must integrate it with an endpoint in the backend. A backend endpoint is also referred to as an integration endpoint and can be a Lambda function, an HTTP webpage, or an AWS service action.

As with the API method, the API integration has an integration request and an integration response. An integration request encapsulates an HTTP request received by the backend. It might or might not differ from the method request submitted by the client. An integration response is an HTTP response encapsulating the output returned by the backend.

Setting up an integration request involves the following: configuring how to pass client-submitted method requests to the backend; configuring how to transform the request data, if necessary, to the integration request data; and specifying which Lambda function to call, specifying which HTTP server to forward the incoming request to, or specifying the AWS service action to invoke.

Setting up an integration response (applicable to non-proxy integrations only) involves the following: configuring how to pass the backend-returned result to a method response of a given status code, configuring how to transform specified integration response parameters to preconfigured method response parameters, and configuring how to map the integration response body to the method response body according to the specified body-mapping templates.

Programmatically, an integration request is encapsulated by the `Integration` resource and an integration response by the `IntegrationResponse` resource of API Gateway.

To set up an integration request, you create an `Integration` resource and use it to configure the integration endpoint URL. You then set the IAM permissions to access the backend, and specify mappings to transform the incoming request data before passing it to the backend. To set up an integration response for non-proxy integration, you create an `IntegrationResponse` resource and use it to set its target method response. You then configure how to map backend output to the method response.

Topics

Set up an integration request in API Gateway

To set up an integration request, you perform the following required and optional tasks:

1. Choose an integration type that determines how method request data is passed to the backend.

2. For non-mock integrations, specify an HTTP method and the URI of the targeted integration endpoint, except for the MOCK integration.
3. For integrations with Lambda functions and other AWS service actions, set an IAM role with required permissions for API Gateway to call the backend on your behalf.
4. For non-proxy integrations, set necessary parameter mappings to map predefined method request parameters to appropriate integration request parameters.
5. For non-proxy integrations, set necessary body mappings to map the incoming method request body of a given content type according to the specified mapping template.
6. For non-proxy integrations, specify the condition under which the incoming method request data is passed through to the backend as-is.
7. Optionally, specify how to handle type conversion for a binary payload.
8. Optionally, declare a cache namespace name and cache key parameters to enable API caching.

Performing these tasks involves creating an Integration resource of API Gateway and setting appropriate property values. You can do so using the API Gateway console, AWS CLI commands, an AWS SDK, or the API Gateway REST API.

Topics

- Basic tasks of an API integration request
- Choose an API Gateway API integration type
- Set up a proxy integration with a proxy resource
- Set up an API integration request using the API Gateway console

Basic tasks of an API integration request

An integration request is an HTTP request that API Gateway submits to the backend, passing along the client-submitted request data, and transforming the data, if necessary. The HTTP method (or verb) and URI of the integration request are dictated by the backend (that is, the integration endpoint). They can be the same as or different from the method request's HTTP method and URI, respectively.

For example, when a Lambda function returns a file that is fetched from Amazon S3, you can expose this operation intuitively as a GET method request to the client even though the corresponding integration request requires that a POST request be used to invoke the Lambda function. For an HTTP endpoint, it is likely that the method request and the corresponding integration request both use the same HTTP verb. However, this is not required. You can integrate the following method request:

```
GET /{var}?query=value
Host: api.domain.net
```

With the following integration request:

```
POST /
Host: service.domain.com
Content-Type: application/json
Content-Length: ...

{
   path: "{var}'s value",
   type: "value"
}
```

As an API developer, you can use whatever HTTP verb and URI for a method request suit your requirements. But you must follow the requirements of the integration endpoint. When the method request data differs from the integration request data, you can reconcile the difference by providing mappings from the method request data to the integration request data.

In the preceding examples, the mapping translates the path variable (`{var}`) and the query parameter (`query`) values of the `GET` method request to the values of the integration request's payload properties of `path` and `type`. Other mappable request data includes request headers and body. These are described in [Set up request and response data mappings using the API Gateway console](#).

When setting up the HTTP or HTTP proxy integration request, you assign the backend HTTP endpoint URL as the integration request URI value. For example, in the PetStore API, the method request to get a page of pets has the following integration request URI:

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets
```

When setting up the Lambda or Lambda proxy integration, you assign the Amazon Resource Name (ARN) for invoking the Lambda function as the integration request URI value. This ARN has the following format:

```
arn:aws:apigateway:api-region:lambda:path//2015-03-31/functions/arn:aws:lambda:lambda-region:account-id:function:lambda-function-name/invocations
```

The part after `arn:aws:apigateway:api-region:lambda:path/`, namely, `/2015-03-31/functions/arn:aws:lambda:lambda-region:account-id:function:lambda-function-name/invocations`, is the REST API URI path of the Lambda Invoke action. If you use the API Gateway console to set up the Lambda integration, API Gateway creates the ARN and assigns it to the integration URI after prompting you to choose the `lambda-function-name` from a region.

When setting up the integration request with another AWS service action, the integration request URI is also an ARN, similar to the integration with the Lambda `Invoke` action. For example, for the integration with the GetBucket action of Amazon S3, the integration request URI is an ARN of the following format:

```
arn:aws:apigateway:api-region:s3:path/{bucket}
```

The integration request URI is of the path convention to specify the action, where `{bucket}` is the placeholder of a bucket name. Alternatively, an AWS service action can be referenced by its name. Using the action name, the integration request URI for the `GetBucket` action of Amazon S3 becomes the following:

```
arn:aws:apigateway:api-region:s3:action/GetBucket
```

With the action-based integration request URI, the bucket name (`{bucket}`) must be specified in the integration request body (`{ Bucket: "{bucket}" }`), following the input format of `GetBucket` action.

For AWS integrations, you must also configure credentials to allow API Gateway to call the integrated actions. You can create a new or choose an existing IAM role for API Gateway to call the action and then specify the role using its ARN. The following shows an example of this ARN:

```
arn:aws:iam::account-id:role/iam-role-name
```

This IAM role must contain a policy to allow the action to be executed. It must also have API Gateway declared (in the role's trust relationship) as a trusted entity to assume the role. Such permissions can be granted on the action itself. They are known as resource-based permissions. For the Lambda integration, you can call the Lambda's addPermission action to set the resource-based permissions and then set `credentials` to null in the API Gateway integration request.

We discussed the basic integration setup. Advanced settings involve mapping method request data to the integration request data. After discussing the basic setup for an integration response, we cover advanced topics in Set up request and response data mappings using the API Gateway console, where we also cover passing payload through and handling content encodings.

Choose an API Gateway API integration type

You choose an API integration type according to the types of integration endpoint you work with and how you want data to pass to and from the integration endpoint. For a Lambda function, you can have the Lambda proxy integration, or the Lambda custom integration. For an HTTP endpoint, you can have the HTTP proxy integration or the HTTP custom integration. For an AWS service action, you have the AWS integration of the non-proxy type only. API Gateway also supports the mock integration, where API Gateway serves as an integration endpoint to respond to a method request.

The Lambda custom integration is a special case of the AWS integration, where the integration endpoint corresponds to the function-invoking action of the Lambda service.

Programmatically, you choose an integration type by setting the `type` property on the `Integration` resource. For the Lambda proxy integration, the value is `AWS_PROXY`. For the Lambda custom integration and all other AWS integrations, it is `AWS`. For the HTTP proxy integration and HTTP integration, the value is `HTTP_PROXY` and `HTTP`, respectively. For the mock integration, the `type` value is `MOCK`.

The Lambda proxy integration supports a streamlined integration setup with a single Lambda function. The setup is simple and can evolve with the backend without having to tear down the existing setup. For these reasons, it is highly recommended for integration with a Lambda function.

In contrast, the Lambda custom integration allows for reuse of configured mapping templates for various integration endpoints that have similar requirements of the input and output data formats. The setup is more involved and is recommended for more advanced application scenarios.

Similarly, the HTTP proxy integration has a streamlined integration setup and can evolve with the backend without having to tear down the existing setup. The HTTP custom integration is more involved to set up, but allows for reuse of configured mapping templates for other integration endpoints.

The following list summarizes the supported integration types:

- AWS: This type of integration lets an API expose AWS service actions. In AWS integration, you must configure both the integration request and integration response and set up necessary data mappings from the method request to the integration request, and from the integration response to the method response.
- AWS_PROXY: This type of integration lets an API method be integrated with the Lambda function invocation action with a flexible, versatile, and streamlined integration setup. This integration relies on direct interactions between the client and the integrated Lambda function.

  With this type of integration, also known as the Lambda proxy integration, you do not set the integration request or the integration response. API Gateway passes the incoming request from the client as the input to the backend Lambda function. The integrated Lambda function takes the input of this format and parses the input from all available sources, including request headers, URL path variables, query string parameters, and applicable body. The function returns the result following this output format.

  This is the preferred integration type to call a Lambda function through API Gateway and is not applicable to any other AWS service actions, including Lambda actions other than the function-invoking action.
- HTTP: This type of integration lets an API expose HTTP endpoints in the backend. With the HTTP integration, also known as the HTTP custom integration, you must configure both the integration request and integration response. You must set up necessary data mappings from the method request to the integration request, and from the integration response to the method response.
- HTTP_PROXY: The HTTP proxy integration allows a client to access the backend HTTP endpoints with a streamlined integration setup on single API method. You do not set the integration request or the integration response. API Gateway passes the incoming request from the client to the HTTP endpoint and passes the outgoing response from the HTTP endpoint to the client.
- MOCK: This type of integration lets API Gateway return a response without sending the request further to the backend. This is useful for API testing because it can be used to test the integration set up without incurring charges for using the backend and to enable collaborative development of an API.

  In collaborative development, a team can isolate their development effort by setting up simulations of API components owned by other teams by using the MOCK integrations. It is also used to return CORS-related headers to ensure that the API method permits CORS access. In fact, the API Gateway console integrates the OPTIONS method to support CORS with a mock integration. Gateway responses are other examples of mock integrations.

Set up a proxy integration with a proxy resource

To set up a proxy integration in an API Gateway API with a proxy resource, you perform the following tasks:

- Create a proxy resource with a greedy path variable of `{proxy+}`.
- Set the `ANY` method on the proxy resource.
- Integrate the resource and method with a backend using the HTTP or Lambda integration type.

> **Note**
>
> Greedy path variables, `ANY` methods, and proxy integration types are independent features, although they are commonly used together. You can configure a specific HTTP method on a greedy resource or apply non-proxy integration types to a proxy resource.

API Gateway enacts certain restrictions and limitations when handling methods with either a Lambda proxy integration or an HTTP proxy integration. For details, see the section called "Important notes".

> **Note**
>
> When using proxy integration with a passthrough, API Gateway returns the default `Content-Type:application/json` header if the content type of a payload is unspecified.

A proxy resource is most powerful when it is integrated with a backend using either HTTP proxy integration or Lambda proxy integration.

HTTP proxy integration with a proxy resource

The HTTP proxy integration, designated by `HTTP_PROXY` in the API Gateway REST API, is for integrating a method request with a backend HTTP endpoint. With this integration type, API Gateway simply passes the entire request and response between the frontend and the backend, subject to certain restrictions and limitations.

Note

HTTP proxy integration supports multi-valued headers and query strings.

When applying the HTTP proxy integration to a proxy resource, you can set up your API to expose a portion or an entire endpoint hierarchy of the HTTP backend with a single integration setup. For example, suppose the backend of the website is organized into multiple branches of tree nodes off the root node (`/site`) as: `/site/a`$_0$`/a`$_1$`/.../a`$_N$, `/site/b`$_0$`/b`$_1$`/.../b`$_M$, etc. If you integrate the `ANY` method on a proxy resource of `/api/{proxy+}` with the backend endpoints with URL paths of `/site/{proxy}`, a single integration request can support any HTTP operations (GET, POST, etc.) on any of `[a`$_0$`, a`$_1$`, ..., a`$_N$`, b`$_0$`, b`$_1$`, ...b`$_M$`, ...]`. If you apply a proxy integration to a specific HTTP method, for example, `GET`, instead, the resulting integration request works with the specified (that is, `GET`) operations on any of those backend nodes.

Lambda proxy integration with a proxy resource

The Lambda proxy integration, designated by `AWS_PROXY` in the API Gateway REST API, is for integrating a method request with a Lambda function in the backend. With this integration type, API Gateway applies a default mapping template to send the entire request to the Lambda function and transforms the output from the Lambda function to HTTP responses.

Similarly, you can apply the Lambda proxy integration to a proxy resource of `/api/{proxy+}` to set up a single integration to have a backend Lambda function react individually to changes in any of the API resources under `/api`.

Set up an API integration request using the API Gateway console

An API method setup defines the method and describes its behaviors. To set up a method, you must specify a resource, including the root ("/"), on which the method is exposed, an HTTP method (`GET`, `POST`, etc.), and how it will be integrated with the targeted backend. The method request and response specify the contract with the calling app, stipulating which parameters the API can receive and what the response looks like.

The following procedures describe how to use the API Gateway console to create an integration request.

Topics

- Set up a Lambda integration

- Set up an HTTP integration
- Set up an AWS service integration
- Set up a mock integration

Set up a Lambda integration

Use a Lambda function integration to integrate your API with a Lambda function. At the API level, this is an `AWS` integration type if you create a non-proxy integration, or an `AWS_PROXY` integration type if you create a proxy integration.

To set up a Lambda integration

1. In the **Resources** pane, choose **Create method**.
2. For **Method type**, select an HTTP method.
3. For **Integration type**, choose **Lambda function**.
4. To use a Lambda proxy integration, turn on **Lambda proxy integration**. To learn more about Lambda proxy integrations, see the section called " Understand Lambda proxy integration ".
5. For **Lambda function**, enter the name of the Lambda function.

   If you are using a Lambda function in a different Region than your API, select the Region from the dropdown menu and enter the name of the Lambda function. If you are using a cross-account Lambda function, enter the function ARN.
6. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.
7. (Optional) You can configure the method request settings using the following dropdown menus. Choose **Method request settings** and configure your method request. For more information, see step 3 of the section called "Edit a method request in the console".

   You can also configure your method request settings after you create your method.
8. Choose **Create method**.

Set up an HTTP integration

Use an HTTP integration to integrate your API with an HTTP endpoint. At the API level, this is the HTTP integration type.

To set up an HTTP integration

1. In the **Resources** pane, choose **Create method**.
2. For **Method type**, select an HTTP method.
3. For **Integration type**, choose **HTTP**.
4. To use an HTTP proxy integration, turn on **HTTP proxy integration**. To learn more about HTTP proxy integrations, see the section called “Set up HTTP proxy integrations in API Gateway”.
5. For **HTTP method**, choose the HTTP method type that most closely matches the method in the HTTP backend.
6. For **Endpoint URL**, enter the URL of the HTTP backend you want this method to use.
7. For **Content handling**, select a content handling behavior.
8. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.
9. (Optional) You can configure the method request settings using the following dropdown menus. Choose **Method request settings** and configure your method request. For more information, see step 3 of the section called “Edit a method request in the console”.

   You can also configure your method request settings after you create your method.
10. Choose **Create method**.

Set up an AWS service integration

Use an AWS service integration to integrate your API directly with an AWS service. At the API level, this is the `AWS` integration type.

To set up an API Gateway API to do any of the following:

- Create a new Lambda function.
- Set a resource permission on the Lambda function.
- Perform any other Lambda service actions.

You must choose **AWS service**.

To set up an AWS service integration

1. In the **Resources** pane, choose **Create method**.
2. For **Method type**, select an HTTP method.
3. For **Integration type**, choose **AWS service**.
4. For **AWS Region**, choose the AWS Region you want this method to use to call the action.
5. For **AWS service**, choose the AWS service you want this method to call.
6. For **AWS subdomain**, enter the subdomain used by the AWS service. Typically, you would leave this blank. Some AWS services can support subdomains as part of the hosts. Consult the service documentation for the availability and, if available, details.
7. For **HTTP method**, choose the HTTP method type that corresponds to the action. For HTTP method type, see the API reference documentation for the AWS service you chose for **AWS service**.
8. For **Action type**, select to either **Use action name** to use an API action or **Use path override** to use a custom resource path. For available actions and custom resource paths, see the API reference documentation for the AWS service you chose for **AWS service**.
9. Enter either an **Action name** or **Path override**.
10. For **Execution role**, enter the ARN of the IAM role that the method will use to call the action.

    To create the IAM role, you can adapt the instructions in the section called “Step 1: Create the AWS service proxy execution role”. Specify an access policy of the following format, with the desired number of action and resource statements:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "action-statement"
      ],
      "Resource": [
        "resource-statement"
      ]
    },
    ...
  ]
}
```

For the action and resource statement syntax, see the documentation for the AWS service you chose for **AWS service**.

For the IAM role's trust relationship, specify the following, which enables API Gateway to take action on behalf of your AWS account:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

11. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.
12. (Optional) You can configure the method request settings using the following dropdown menus. Choose **Method request settings** and configure your method request. For more information, see step 3 of the section called “Edit a method request in the console”.

    You can also configure your method request settings after you create your method.
13. Choose **Create method**.

Set up a mock integration

Use a mock integration if you want API Gateway to act as your backend to return static responses. At the API level, this is the `MOCK` integration type. Typically, you can use the `MOCK` integration when your API is not yet final, but you want to generate API responses to unblock dependent teams for testing. For the `OPTION` method, API Gateway sets the `MOCK` integration as default to return CORS-enabling headers for the applied API resource.

To set up a mock integration

1. In the **Resources** pane, choose **Create method**.
2. For **Method type**, select an HTTP method.
3. For **Integration type**, choose **Mock**.
4. (Optional) You can configure the method request settings using the following dropdown menus. Choose **Method request settings** and configure your method request. For more information, see step 3 of the section called "Edit a method request in the console".

   You can also configure your method request settings after you create your method.
5. Choose **Create method**.

Set up an integration response in API Gateway

For a non-proxy integration, you must set up at least one integration response, and make it the default response, to pass the result returned from the backend to the client. You can choose to pass through the result as-is or to transform the integration response data to the method response data if the two have different formats.

For a proxy integration, API Gateway automatically passes the backend output to the client as an HTTP response. You do not set either an integration response or a method response.

To set up an integration response, you perform the following required and optional tasks:

1. Specify an HTTP status code of a method response to which the integration response data is mapped. This is required.
2. Define a regular expression to select backend output to be represented by this integration response. If you leave this empty, the response is the default response that is used to catch any response not yet configured.
3. If needed, declare mappings consisting of key-value pairs to map specified integration response parameters to given method response parameters.
4. If needed, add body-mapping templates to transform given integration response payloads into specified method response payloads.
5. If needed, specify how to handle type conversion for a binary payload.

An integration response is an HTTP response encapsulating the backend response. For an HTTP endpoint, the backend response is an HTTP response. The integration response status code can take the backend-returned status code, and the integration response body is the backend-returned payload. For a Lambda endpoint, the backend response is the output returned from the Lambda function. With the Lambda integration, the Lambda function output is returned as a `200 OK` response. The payload can contain the result as JSON data, including a JSON string or a JSON object, or an error message as a JSON object. You can assign a regular expression to the selectionPattern property to map an error response to an appropriate HTTP error response. For more information about the Lambda function error response, see Handle Lambda errors in API Gateway. With the Lambda proxy integration, the Lambda function must return output of the following format:

```
{
    statusCode: "...",            // a valid HTTP status code
    headers: {
        custom-header: "..."      // any API-specific custom header
    },
    body: "...",                  // a JSON string.
    isBase64Encoded:  true|false  // for binary support
}
```

There is no need to map the Lambda function response to its proper HTTP response.

To return the result to the client, set up the integration response to pass the endpoint response through as-is to the corresponding method response. Or you can map the endpoint response data to the method response data. The response data that can be mapped includes the response status code, response header parameters, and response body. If no method response is defined for the returned status code, API Gateway returns a 500 error. For more information, see Use a mapping template to override an API's request and response parameters and status codes.

Set up Lambda integrations in API Gateway

You can integrate an API method with a Lambda function using Lambda proxy integration or Lambda non-proxy (custom) integration.

In Lambda proxy integration, the required setup is simple. Set the integration's HTTP method to POST, the integration endpoint URI to the ARN of the Lambda function invocation action of a specific Lambda function, and grant API Gateway permission to call the Lambda function on your behalf.

In Lambda non-proxy integration, in addition to the proxy integration setup steps, you also specify how the incoming request data is mapped to the integration request and how the resulting integration response data is mapped to the method response.

Topics

Set up Lambda proxy integrations in API Gateway

Topics

Understand API Gateway Lambda proxy integration

Amazon API Gateway Lambda proxy integration is a simple, powerful, and nimble mechanism to build an API with a setup of a single API method. The Lambda proxy integration allows the client to call a single Lambda function in the backend. The function accesses many resources or features of other AWS services, including calling other Lambda functions.

In Lambda proxy integration, when a client submits an API request, API Gateway passes to the integrated Lambda function an event object, except that the order of the request parameters is not preserved. This request data includes the request headers, query string parameters, URL path variables, payload, and API configuration data. The configuration data can include current deployment stage name, stage variables, user identity, or authorization context (if any). The backend Lambda function parses the incoming request data to determine the response that it returns. For API Gateway to pass the Lambda output as the API response to the client, the Lambda function must return the result in this format.

Because API Gateway doesn't intervene very much between the client and the backend Lambda function for the Lambda proxy integration, the client and the integrated Lambda function can adapt to changes in each other without breaking the existing integration setup of the API. To enable this, the client must follow application protocols enacted by the backend Lambda function.

You can set up a Lambda proxy integration for any API method. But a Lambda proxy integration is more potent when it is configured for an API method involving a generic proxy resource. The generic proxy resource can be denoted by a special templated path variable of `{proxy+}`, the catch-all `ANY` method placeholder, or both. The client can pass the input to the backend Lambda function in the incoming request as request parameters or applicable payload. The request parameters include headers, URL path variables, query string parameters, and the applicable payload. The integrated Lambda function verifies all of the input sources before processing the request and responding to the client with meaningful error messages if any of the required input is missing.

When calling an API method integrated with the generic HTTP method of `ANY` and the generic resource of `{proxy+}`, the client submits a request with a particular HTTP method in place of `ANY`. The client also specifies a particular URL path instead of `{proxy+}`, and includes any required headers, query string parameters, or an applicable payload.

The following list summarizes runtime behaviors of different API methods with the Lambda proxy integration:

- `ANY /{proxy+}`: The client must choose a particular HTTP method, must set a particular resource path hierarchy, and can set any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.
- `ANY /res`: The client must choose a particular HTTP method and can set any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.
- `GET|POST|PUT|... /{proxy+}`: The client can set a particular resource path hierarchy, any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.
- `GET|POST|PUT|... /res/{path}/...`: The client must choose a particular path segment (for the `{path}` variable) and can set any request headers, query string parameters, and applicable payload to pass input data to the integrated Lambda function.
- `GET|POST|PUT|... /res`: The client can choose any request headers, query string parameters, and applicable payload to pass input data to the integrated Lambda function.

Both the proxy resource of `{proxy+}` and the custom resource of `{custom}` are expressed as templated path variables. However `{proxy+}` can refer to any resource along a path hierarchy, while `{custom}` refers to a particular path segment only. For example, a grocery store might organize its online product inventory by department names, produce categories, and product types. The grocery store's website can then represent available products by the following templated path variables of custom resources: `/{department}/{produce-category}/{product-type}`. For example, apples are represented by `/produce/fruit/apple` and carrots by `/produce/vegetables/carrot`. It can also use `/{proxy+}` to represent any department, any produce category, or any product type that a customer can search for while shopping in the online store. For example, `/{proxy+}` can refer to any of the following items:

- `/produce`
- `/produce/fruit`
- `/produce/vegetables/carrot`

To let customers search for any available product, its produce category, and the associated store department, you can expose a single method of `GET /{proxy+}` with read-only permissions. Similarly, to allow a supervisor to update the `produce` department's inventory, you can set up another single method of `PUT /produce/{proxy+}` with read/write permissions. To allow a cashier to update the running total of a vegetable, you can set up a `POST /produce/vegetables/{proxy+}` method with read/write permissions. To let a store manager perform any possible action on any available product, the online store developer can expose the `ANY /{proxy+}` method with read/write permissions. In any case, at run time, the customer or the employee must select a particular product of a given type in a chosen department, a specific produce category in a chosen department, or a specific department.

For more information about setting up API Gateway proxy integrations, see Set up a proxy integration with a proxy resource.

Proxy integration requires that the client have more detailed knowledge of the backend requirements. Therefore, to ensure optimal app performance and user experience, the backend developer must communicate clearly to the client developer the requirements of the backend, and provide a robust error feedback mechanism when the requirements are not met.

Support for multi-value headers and query string parameters

API Gateway supports multiple headers and query string parameters that have the same name. Multi-value headers as well as single-value headers and parameters can be combined in the same requests and responses. For more information, see Input format of a Lambda function for proxy integration and Output format of a Lambda function for proxy integration.

Set up a proxy resource with Lambda proxy integration

To set up a proxy resource with the Lambda proxy integration type, create an API resource with a greedy path parameter (for example, `/parent/{proxy+}`) and integrate this resource with a Lambda function backend (for example, `arn:aws:lambda:us-west-2:123456789012:function:SimpleLambda4ProxyResource`) on the `ANY` method. The greedy path parameter must be at the end of the API resource path. As with a non-proxy resource, you can set up the proxy resource by using the API Gateway console, importing an OpenAPI definition file, or calling the API Gateway REST API directly.

The following OpenAPI API definition file shows an example of an API with a proxy resource that is integrated with a Lambda function named `SimpleLambda4ProxyResource`.

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-09-12T17:50:37Z",
      "title": "ProxyIntegrationWithLambda"
   },
   "paths": {
      "/{proxy+}": {
         "x-amazon-apigateway-any-method": {
            "parameters": [
               {
                  "name": "proxy",
                  "in": "path",
                  "required": true,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {},
            "x-amazon-apigateway-integration": {
               "responses": {
                  "default": {
                     "statusCode": "200"
                  }
               },
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:SimpleLambda4ProxyResource/
invocations",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "POST",
               "cacheNamespace": "roq9wj",
               "cacheKeyParameters": [
                  "method.request.path.proxy"
               ],
               "type": "aws_proxy"
            }
         }
      }
   },
   "servers": [
      {
         "url": "https://gy415nuibc.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/testStage"
            }
         }
      }
   ]
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-12T17:50:37Z",
    "title": "ProxyIntegrationWithLambda"
  },
  "host": "gy415nuibc.execute-api.us-east-1.amazonaws.com",
  "basePath": "/testStage",
  "schemes": [
    "https"
  ],
  "paths": {
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:SimpleLambda4ProxyResource/
invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "cacheNamespace": "roq9wj",
          "cacheKeyParameters": [
            "method.request.path.proxy"
          ],
          "type": "aws_proxy"
        }
      }
    }
  }
}
```

In Lambda proxy integration, at run time, API Gateway maps an incoming request into the input `event` parameter of the Lambda function. The input includes the request method, path, headers, any query string parameters, any payload, associated context, and any defined stage variables.

The input format is explained in Input format of a Lambda function for proxy integration. For API Gateway to map the Lambda output to HTTP responses successfully, the Lambda function must output the result in the format described in Output format of a Lambda function for proxy integration.

In Lambda proxy integration of a proxy resource through the ANY method, the single backend Lambda function serves as the event handler for all requests through the proxy resource. For example, to log traffic patterns, you can have a mobile device send its location information of state, city, street, and building by submitting a request with /state/city/street/house in the URL path for the proxy resource. The backend Lambda function can then parse the URL path and insert the location tuples into a DynamoDB table.

Set up Lambda proxy integration using the AWS CLI

In this section, we show how to use AWS CLI to set up an API with the Lambda proxy integration.

> **Note**
>
> For detailed instructions for using the API Gateway console to configure a proxy resource with the Lambda proxy integration, see Tutorial: Build a Hello World REST API with Lambda proxy integration.

As an example, we use the following sample Lambda function as the backend of the API:

```
export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    var greeter = 'World';
    if (event.greeter && event.greeter!=="") {
        greeter =  event.greeter;
    } else if (event.body && event.body !== "") {
        var body = JSON.parse(event.body);
        if (body.greeter && body.greeter !== "") {
            greeter = body.greeter;
        }
    } else if (event.queryStringParameters && event.queryStringParameters.greeter &&
 event.queryStringParameters.greeter !== "") {
        greeter = event.queryStringParameters.greeter;
    } else if (event.multiValueHeaders && event.multiValueHeaders.greeter &&
 event.multiValueHeaders.greeter != "") {
        greeter = event.multiValueHeaders.greeter.join(" and ");
    } else if (event.headers && event.headers.greeter && event.headers.greeter != "") {
        greeter = event.headers.greeter;
    }

    res.body = "Hello, " + greeter + "!";
    callback(null, res);
};
```

Comparing this to the Lambda custom integration setup, the input to this Lambda function can be expressed in the request parameters and body. You have more latitude to allow the client to pass the same input data. Here, the client can pass the greeter's name in as a query string parameter, a header, or a body property. The function can also support the Lambda custom integration. The API setup is simpler. You do not configure the method response or integration response at all.

To set up a Lambda proxy integration using the AWS CLI

1. Call the `create-rest-api` command to create an API:

   ```
   aws apigateway create-rest-api --name 'HelloWorld (AWS CLI)' --region us-west-2
   ```

   Note the resulting API's `id` value (`te6si5ach7`) in the response:

   ```
   {
       "name": "HelloWorldProxy (AWS CLI)",
       "id": "te6si5ach7",
       "createdDate": 1508461860
   }
   ```

   You need the API `id` throughout this section.

2. Call the `get-resources` command to get the root resource `id`:

   ```
   aws apigateway get-resources --rest-api-id te6si5ach7 --region us-west-2
   ```

   The successful response is shown as follows:

```
{
    "items": [
        {
            "path": "/",
            "id": "krznpq9xpg"
        }
    ]
}
```

Note the root resource `id` value (`krznpq9xpg`). You need it in the next step and later.

3. Call `create-resource` to create an API Gateway [Resource](#) of `/greeting`:

```
aws apigateway create-resource --rest-api-id te6si5ach7 \
      --region us-west-2 \
      --parent-id krznpq9xpg \
      --path-part {proxy+}
```

The successful response is similar to the following:

```
{
    "path": "/{proxy+}",
    "pathPart": "{proxy+}",
    "id": "2jf6xt",
    "parentId": "krznpq9xpg"
}
```

Note the resulting `{proxy+}` resource's `id` value (`2jf6xt`). You need it to create a method on the `/{proxy+}` resource in the next step.

4. Call `put-method` to create an `ANY` method request of `ANY /{proxy+}`:

```
aws apigateway put-method --rest-api-id te6si5ach7 \
       --region us-west-2 \
       --resource-id 2jf6xt \
       --http-method ANY \
       --authorization-type "NONE"
```

The successful response is similar to the following:

```
{
    "apiKeyRequired": false,
    "httpMethod": "ANY",
    "authorizationType": "NONE"
}
```

This API method allows the client to receive or send greetings from the Lambda function at the backend.

5. Call `put-integration` to set up the integration of the `ANY /{proxy+}` method with a Lambda function, named `HelloWorld`. This function responds to the request with a message of `"Hello, {name}!"`, if the `greeter` parameter is provided, or `"Hello, World!"`, if the query string parameter is not set.

```
aws apigateway put-integration \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method ANY \
        --type AWS_PROXY \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:HelloWorld/invocations \
        --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole
```

> ⚠ **Important**
>
> For Lambda integrations, you must use the HTTP method of `POST` for the integration request, according to the specification of the Lambda service action for function invocations. The IAM role of `apigAwsProxyRole` must have policies allowing the `apigateway` service to invoke Lambda functions. For more information about IAM permissions, see the section called " API Gateway permissions model for invoking an API".

The successful output is similar to the following:

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:1234567890:function:HelloWorld/invocations",
    "httpMethod": "POST",
    "cacheNamespace": "vvom7n",
    "credentials": "arn:aws:iam::1234567890:role/apigAwsProxyRole",
    "type": "AWS_PROXY"
}
```

Instead of supplying an IAM role for `credentials`, you can call the add-permission command to add resource-based permissions. This is what the API Gateway console does.

6. Call `create-deployment` to deploy the API to a `test` stage:

```
aws apigateway create-deployment --rest-api-id te6si5ach7 --stage-name test --
region us-west-2
```

7. Test the API using the following cURL commands in a terminal.

Calling the API with the query string parameter of `?greeter=jane`:

```
curl -X GET 'https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/greeting?
greeter=jane'
```

Calling the API with a header parameter of `greeter:jane`:

```
curl -X GET https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/hi \
  -H 'content-type: application/json' \
  -H 'greeter: jane'
```

Calling the API with a body of `{"greeter":"jane"}`:

```
curl -X POST https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/hi \
  -H 'content-type: application/json' \
  -d '{ "greeter": "jane" }'
```

In all the cases, the output is a 200 response with the following response body:

```
Hello, jane!
```

Input format of a Lambda function for proxy integration

In Lambda proxy integration, API Gateway maps the entire client request to the input `event` parameter of the backend Lambda function. The following example shows the structure of an event that API Gateway sends to a Lambda proxy integration.

```
{
  "resource": "/my/path",
  "path": "/my/path",
  "httpMethod": "GET",
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "multiValueHeaders": {
    "header1": [
      "value1"
    ],
    "header2": [
      "value1",
      "value2"
    ]
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "multiValueQueryStringParameters": {
    "parameter1": [
      "value1",
      "value2"
    ],
    "parameter2": [
      "value"
    ]
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "id",
    "authorizer": {
      "claims": null,
      "scopes": null
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "extendedRequestId": "request-id",
    "httpMethod": "GET",
    "identity": {
      "accessKey": null,
      "accountId": null,
      "caller": null,
      "cognitoAuthenticationProvider": null,
      "cognitoAuthenticationType": null,
      "cognitoIdentityId": null,
      "cognitoIdentityPoolId": null,
      "principalOrgId": null,
      "sourceIp": "IP",
      "user": null,
      "userAgent": "user-agent",
      "userArn": null,
      "clientCert": {
       "clientCertPem": "CERT_CONTENT",
       "subjectDN": "www.example.com",
       "issuerDN": "Example issuer",
       "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
       "validity": {
        "notBefore": "May 28 12:30:02 2019 GMT",
        "notAfter": "Aug  5 09:36:04 2021 GMT"
       }
      }
    },
    "path": "/my/path",
    "protocol": "HTTP/1.1",
    "requestId": "id=",
    "requestTime": "04/Mar/2020:19:15:17 +0000",
    "requestTimeEpoch": 1583349317135,
    "resourceId": null,
    "resourcePath": "/my/path",
    "stage": "$default"
  },
  "pathParameters": null,
  "stageVariables": null,
  "body": "Hello from Lambda!",
  "isBase64Encoded": false
}
```

Note

In the input:

- The `headers` key can only contain single-value headers.
- The `multiValueHeaders` key can contain multi-value headers as well as single-value headers.
- If you specify values for both `headers` and `multiValueHeaders`, API Gateway merges them into a single list. If the same key-value pair is specified in both, only the values from `multiValueHeaders` will appear in the merged list.

In the input to the backend Lambda function, the `requestContext` object is a map of key-value pairs. In each pair, the key is the name of a $context variable property, and the value is the value of that property. API Gateway may add new keys to the map.

Depending on the features that are enabled, the `requestContext` map may vary from API to API. For example, in the preceding example, no authorization type is specified, so no `$context.authorizer.*` or `$context.identity.*` properties are present. When an authorization type is specified, this causes API Gateway to pass authorized user information to the integration endpoint in a `requestContext.identity` object as follows:

- When the authorization type is `AWS_IAM`, the authorized user information includes `$context.identity.*` properties.
- When the authorization type is `COGNITO_USER_POOLS` (Amazon Cognito authorizer), the authorized user information includes `$context.identity.cognito*` and `$context.authorizer.claims.*` properties.
- When the authorization type is `CUSTOM` (Lambda authorizer), the authorized user information includes `$context.authorizer.principalId` and other applicable `$context.authorizer.*` properties.

Output format of a Lambda function for proxy integration

In Lambda proxy integration, API Gateway requires the backend Lambda function to return output according to the following JSON format:

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headerName": "headerValue", ... },
    "multiValueHeaders": { "headerName": ["headerValue", "headerValue2", ...], ... },
    "body": "..."
}
```

In the output:

- The `headers` and `multiValueHeaders` keys can be unspecified if no extra response headers are to be returned.
- The `headers` key can only contain single-value headers.
- The `multiValueHeaders` key can contain multi-value headers as well as single-value headers. You can use the `multiValueHeaders` key to specify all of your extra headers, including any single-value ones.
- If you specify values for both `headers` and `multiValueHeaders`, API Gateway merges them into a single list. If the same key-value pair is specified in both, only the values from `multiValueHeaders` will appear in the merged list.

To enable CORS for the Lambda proxy integration, you must add `Access-Control-Allow-Origin:`*`domain-name`* to the output `headers`. *`domain-name`* can be `*` for any domain name. The output `body` is marshalled to the frontend as the method response payload. If `body` is a binary blob, you can encode it as a Base64-encoded string by setting `isBase64Encoded` to `true` and configuring `*/*` as a **Binary Media Type**. Otherwise, you can set it to `false` or leave it unspecified.

> **Note**
>
> For more information about enabling binary support, see Enabling binary support using the API Gateway console. For an example Lambda function, see Return binary media from a Lambda proxy integration.

If the function output is of a different format, API Gateway returns a `502 Bad Gateway` error response.

To return a response in a Lambda function in Node.js, you can use commands such as the following:

- To return a successful result, call `callback(null, {"statusCode": 200, "body": "results"})`.
- To throw an exception, call `callback(new Error('internal server error'))`.
- For a client-side error (if, for example, a required parameter is missing), you can call `callback(null, {"statusCode": 400, "body": "Missing parameters of ..."})` to return the error without throwing an exception.

In a Lambda `async` function in Node.js, the equivalent syntax would be:

- To return a successful result, call `return {"statusCode": 200, "body": "results"}`.
- To throw an exception, call `throw new Error("internal server error")`.
- For a client-side error (if, for example, a required parameter is missing), you can call `return {"statusCode": 400, "body": "Missing parameters of ..."}` to return the error without throwing an exception.

Set up Lambda custom integrations in API Gateway

To show how to set up the Lambda custom integration, we create an API Gateway API to expose the `GET /greeting?greeter={name}` method to invoke a Lambda function. Use one of the following example Lambda functions for you API.

Use one of the following example Lambda functions:

Node.js

```
export const handler = function(event, context, callback) {
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    if (event.greeter==null) {
        callback(new Error('Missing the required greeter parameter.'));
    } else if (event.greeter === "") {
        res.body = "Hello, World";
        callback(null, res);
    } else {
        res.body = "Hello, " + event.greeter +"!";
        callback(null, res);
    }
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)
    res = {
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    }

    if event['greeter'] == "":
        res['body'] = "Hello, World"
    elif (event['greeter']):
        res['body'] = "Hello, " + event['greeter'] + "!"
    else:
        raise Exception('Missing the required greeter parameter.')

    return res
```

The function responds with a message of "`Hello, {name}!`" if the `greeter` parameter value is a non-empty string. It returns a message of "`Hello, World!`" if the `greeter` value is an empty string. The function returns an error message of "`Missing the required greeter parameter.`" if the greeter parameter is not set in the incoming request. We name the function `HelloWorld`.

You can create it in the Lambda console or by using the AWS CLI. In this section, we reference this function using the following ARN:

```
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld
```

With the Lambda function set in the backend, proceed to set up the API.

To set up the Lambda custom integration using the AWS CLI

1. Call the `create-rest-api` command to create an API:

   ```
   aws apigateway create-rest-api --name 'HelloWorld (AWS CLI)' --region us-west-2
   ```

   Note the resulting API's `id` value (`te6si5ach7`) in the response:

   ```
   {
       "name": "HelloWorld (AWS CLI)",
       "id": "te6si5ach7",
       "createdDate": 1508461860
   }
   ```

   You need the API `id` throughout this section.

2. Call the `get-resources` command to get the root resource `id`:

   ```
   aws apigateway get-resources --rest-api-id te6si5ach7 --region us-west-2
   ```

   The successful response is as follows:

   ```
   {
       "items": [
           {
               "path": "/",
               "id": "krznpq9xpg"
           }
       ]
   }
   ```

   Note the root resource `id` value (`krznpq9xpg`). You need it in the next step and later.

3. Call `create-resource` to create an API Gateway Resource of `/greeting`:

   ```
   aws apigateway create-resource --rest-api-id te6si5ach7 \
         --region us-west-2 \
         --parent-id krznpq9xpg \
         --path-part greeting
   ```

   The successful response is similar to the following:

```
{
    "path": "/greeting",
    "pathPart": "greeting",
    "id": "2jf6xt",
    "parentId": "krznpq9xpg"
}
```

Note the resulting `greeting` resource's `id` value (`2jf6xt`). You need it to create a method on the `/greeting` resource in the next step.

4. Call `put-method` to create an API method request of `GET /greeting?greeter={name}`:

```
aws apigateway put-method --rest-api-id te6si5ach7 \
       --region us-west-2 \
       --resource-id 2jf6xt \
       --http-method GET \
       --authorization-type "NONE" \
       --request-parameters method.request.querystring.greeter=false
```

The successful response is similar to the following:

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "requestParameters": {
        "method.request.querystring.greeter": false
    }
}
```

This API method allows the client to receive a greeting from the Lambda function at the backend. The `greeter` parameter is optional because the backend should handle either an anonymous caller or a self-identified caller.

5. Call `put-method-response` to set up the `200 OK` response to the method request of `GET /greeting?greeter={name}`:

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --status-code 200
```

6. Call `put-integration` to set up the integration of the `GET /greeting?greeter={name}` method with a Lambda function, named `HelloWorld`. The function responds to the request with a message of `"Hello, {name}!"`, if the `greeter` parameter is provided, or `"Hello, World!"`, if the query string parameter is not set.

```
aws apigateway put-integration \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --type AWS \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations \
        --request-templates '{"application/json":"{\"greeter\":
\"$input.params('greeter')\"}"}' \
        --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole
```

The mapping template supplied here translates the `greeter` query string parameter to the `greeter` property of the JSON payload. This is necessary because the input to a Lambda function must be expressed in the body.

Important

For Lambda integrations, you must use the HTTP method of `POST` for the integration request, according to the specification of the Lambda service action for function invocations. The `uri` parameter is the ARN of the function-invoking action. Successful output is similar to the following:

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations",
    "httpMethod": "POST",
    "requestTemplates": {
        "application/json": "{\"greeter\":\"$input.params('greeter')\"}"
    },
    "cacheNamespace": "krznpq9xpg",
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "type": "AWS"
}
```

The IAM role of `apigAwsProxyRole` must have policies that allow the `apigateway` service to invoke Lambda functions. Instead of supplying an IAM role for `credentials`, you can call the add-permission command to add resource-based permissions. This is how the API Gateway console adds these permissions.

7. Call `put-integration-response` to set up the integration response to pass the Lambda function output to the client as the `200 OK` method response.

```
 aws apigateway put-integration-response \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --status-code 200 \
        --selection-pattern ""
```

By setting the selection-pattern to an empty string, the `200 OK` response is the default.

The successful response should be similar to the following:

```
 {
    "selectionPattern": "",
    "statusCode": "200"
}
```

8. Call `create-deployment` to deploy the API to a `test` stage:

```
aws apigateway create-deployment --rest-api-id te6si5ach7 --stage-name test --
region us-west-2
```

9. Test the API using the following cURL command in a terminal:

```
curl -X GET 'https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/greeting?
greeter=me' \
  -H 'authorization: AWS4-HMAC-SHA256 Credential={access_key}/20171020/us-
west-2/execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
 Signature=f327...5751'
```

Set up asynchronous invocation of the backend Lambda function

In Lambda non-proxy (custom) integration, the backend Lambda function is invoked synchronously by default. This is the desired behavior for most REST API operations. Some applications, however, require work to be performed asynchronously (as a batch operation or a long-latency operation), typically by a separate backend component. In this case, the backend Lambda function is invoked asynchronously, and the front-end REST API method doesn't return the result.

You can configure the Lambda function for a Lambda non-proxy integration to be invoked asynchronously by specifying `'Event'` as the Lambda invocation type. This is done as follows:

Configure Lambda asynchronous invocation in the API Gateway console

For all invocations to be asynchronous:

- In **Integration request**, add an `X-Amz-Invocation-Type` header with a static value of `'Event'`.

For clients to decide if invocations are asynchronous or synchronous:

1. In **Method request**, add an `InvocationType` header.
2. In **Integration request** add an `X-Amz-Invocation-Type` header with a mapping expression of `method.request.header.InvocationType`.
3. Clients can include the `InvocationType: Event` header in API requests for asynchronous invocations or `InvocationType: RequestResponse` for synchronous invocations.

Configure Lambda asynchronous invocation using OpenAPI

For all invocations to be asynchronous:

- Add the `X-Amz-Invocation-Type` header to the **x-amazon-apigateway-integration** section.

```
"x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.X-Amz-Invocation-Type" : "'Event'"
          },
          "passthroughBehavior" : "when_no_match",
          "contentHandling" : "CONVERT_TO_TEXT"
        }
```

For clients to decide if invocations are asynchronous or synchronous:

1. Add the following header on any OpenAPI Path Item Object.

```
"parameters" : [ {
"name" : "InvocationType",
"in" : "header",
"schema" : {
  "type" : "string"
}
} ]
```

2. Add the `X-Amz-Invocation-Type` header to **x-amazon-apigateway-integration** section.

```
"x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.X-Amz-Invocation-Type" :
 "method.request.header.InvocationType"
          },
          "passthroughBehavior" : "when_no_match",
          "contentHandling" : "CONVERT_TO_TEXT"
        }
```

3. Clients can include the `InvocationType: Event` header in API requests for asynchronous invocations or `InvocationType: RequestResponse` for synchronous invocations.

Handle Lambda errors in API Gateway

For Lambda custom integrations, you must map errors returned by Lambda in the integration response to standard HTTP error responses for your clients. Otherwise, Lambda errors are returned as `200 OK` responses by default and the result is not intuitive for your API users.

There are two types of errors that Lambda can return: standard errors and custom errors. In your API, you must handle these differently.

With the Lambda proxy integration, Lambda is required to return an output of the following format:

```
{
  "isBase64Encoded" : "boolean",
  "statusCode": "number",
  "headers": { ... },
  "body": "JSON string"
}
```

In this output, `statusCode` is typically `4XX` for a client error and `5XX` for a server error. API Gateway handles these errors by mapping the Lambda error to an HTTP error response, according to the specified `statusCode`. For API Gateway to pass the error type (for example, `InvalidParameterException`), as part of the response to the client, the Lambda function must include a header (for example, `"X-Amzn-ErrorType":"InvalidParameterException"`) in the `headers` property.

Topics

- Handle standard Lambda errors in API Gateway
- Handle custom Lambda errors in API Gateway

Handle standard Lambda errors in API Gateway

A standard AWS Lambda error has the following format:

```
{
  "errorMessage": "<replaceable>string</replaceable>",
  "errorType": "<replaceable>string</replaceable>",
  "stackTrace": [
    "<replaceable>string</replaceable>",
    ...
  ]
}
```

Here, `errorMessage` is a string expression of the error. The `errorType` is a language-dependent error or exception type. The `stackTrace` is a list of string expressions showing the stack trace leading to the occurrence of the error.

For example, consider the following JavaScript (Node.js) Lambda function.

```
export const handler = function(event, context, callback) {
    callback(new Error("Malformed input ..."));
};
```

This function returns the following standard Lambda error, containing `Malformed input ...` as the error message:

```
{
  "errorMessage": "Malformed input ...",
  "errorType": "Error",
  "stackTrace": [
    "export const handler (/var/task/index.js:3:14)"
  ]
}
```

Similarly, consider the following Python Lambda function, which raises an `Exception` with the same `Malformed input ...` error message.

```
def lambda_handler(event, context):
    raise Exception('Malformed input ...')
```

This function returns the following standard Lambda error:

```
{
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      3,
      "lambda_handler",
      "raise Exception('Malformed input ...')"
    ]
  ],
  "errorType": "Exception",
  "errorMessage": "Malformed input ..."
}
```

Note that the `errorType` and `stackTrace` property values are language-dependent. The standard error also applies to any error object that is an extension of the `Error` object or a subclass of the `Exception` class.

To map the standard Lambda error to a method response, you must first decide on an HTTP status code for a given Lambda error. You then set a regular expression pattern on the `selectionPattern` property of the IntegrationResponse associated with the given HTTP status code. In the API Gateway console, this `selectionPattern` is denoted as **Lambda error regex** in the **Integration response** section, under each integration response.

Note

API Gateway uses Java pattern-style regexes for response mapping. For more information, see Pattern in the Oracle documentation.

For example, to set up a new `selectionPattern` expression, using AWS CLI, call the following put-integration-response command:

```
aws apigateway put-integration-response --rest-api-id z0vprf0mdh --resource-id x3o5ih
 --http-method GET --status-code 400 --selection-pattern "Malformed.*" --region us-
west-2
```

Make sure that you also set up the corresponding error code (`400`) on the method response. Otherwise, API Gateway throws an invalid configuration error response at runtime.

Note

At runtime, API Gateway matches the Lambda error's `errorMessage` against the pattern of the regular expression on the `selectionPattern` property. If there is a match, API Gateway returns the Lambda error as an HTTP response of the corresponding HTTP status code. If there is no match, API Gateway returns the error as a default response or throws an invalid configuration exception if no default response is configured.

Setting the `selectionPattern` value to `.*` for a given response amounts to resetting this response as the default response. This is because such a selection pattern will match all error messages, including null, i.e., any unspecified error message. The resulting mapping overrides the default mapping.

To update an existing `selectionPattern` value using the AWS CLI, call the update-integration-response operation to replace the `/selectionPattern` path value with the specified regex expression of the `Malformed*` pattern.

To set the `selectionPattern` expression using the API Gateway console, enter the expression in the **Lambda error regex** text box when setting up or updating an integration response of a specified HTTP status code.

Handle custom Lambda errors in API Gateway

Instead of the standard error described in the preceding section, AWS Lambda allows you to return a custom error object as JSON string. The error can be any valid JSON object. For example, the following JavaScript (Node.js) Lambda function returns a custom error:

```
export const handler = (event, context, callback) => {
    ...
    // Error caught here:
    var myErrorObj = {
        errorType : "InternalServerError",
        httpStatus : 500,
        requestId : context.awsRequestId,
        trace : {
            "function": "abc()",
            "line": 123,
            "file": "abc.js"
        }
    }
    callback(JSON.stringify(myErrorObj));
};
```

You must turn the `myErrorObj` object into a JSON string before calling `callback` to exit the function. Otherwise, the `myErrorObj` is returned as a string of `"[object Object]"`. When a method of your API is integrated with the preceding Lambda function, API Gateway receives an integration response with the following payload:

```
{
    "errorMessage": "{\"errorType\":\"InternalServerError\",\"httpStatus\":500,
\"requestId\":\"e5849002-39a0-11e7-a419-5bb5807c9fb2\",\"trace\":{\"function\":
\"abc()\",\"line\":123,\"file\":\"abc.js\"}}"
}
```

As with any integration response, you can pass through this error response as-is to the method response. Or you can have a mapping template to transform the payload into a different format. For example, consider the following body-mapping template for a method response of `500` status code:

```
{
    errorMessage: $input.path('$.errorMessage');
}
```

This template translates the integration response body that contains the custom error JSON string to the following method response body. This method response body contains the custom error JSON object:

```
{
    "errorMessage" : {
        errorType : "InternalServerError",
        httpStatus : 500,
        requestId : context.awsRequestId,
        trace : {
            "function": "abc()",
            "line": 123,
            "file": "abc.js"
        }
    }
};
```

Depending on your API requirements, you may need to pass some or all of the custom error properties as method response header parameters. You can achieve this by applying the custom error mappings from the integration response body to the method response headers.

For example, the following OpenAPI extension defines a mapping from the `errorMessage.errorType`, `errorMessage.httpStatus`, `errorMessage.trace.function`, and `errorMessage.trace` properties to the `error_type`, `error_status`, `error_trace_function`, and `error_trace` headers, respectively.

```
"x-amazon-apigateway-integration": {
    "responses": {
        "default": {
          "statusCode": "200",
          "responseParameters": {
            "method.response.header.error_trace_function":
 "integration.response.body.errorMessage.trace.function",
            "method.response.header.error_status":
 "integration.response.body.errorMessage.httpStatus",
            "method.response.header.error_type":
 "integration.response.body.errorMessage.errorType",
            "method.response.header.error_trace":
 "integration.response.body.errorMessage.trace"
          },
          ...
        }
    }
}
```

At runtime, API Gateway deserializes the `integration.response.body` parameter when performing header mappings. However, this deserialization applies only to body-to-header mappings for Lambda custom error responses and does not apply to body-to-body mappings using `$input.body`. With these custom-error-body-to-header mappings, the client receives the following headers as part of the method response, provided that the `error_status`, `error_trace`, `error_trace_function`, and `error_type` headers are declared in the method request.

```
"error_status":"500",
"error_trace":"{\"function\":\"abc()\",\"line\":123,\"file\":\"abc.js\"}",
"error_trace_function":"abc()",
"error_type":"InternalServerError"
```

The `errorMessage.trace` property of the integration response body is a complex property. It is mapped to the `error_trace` header as a JSON string.

Set up HTTP integrations in API Gateway

You can integrate an API method with an HTTP endpoint using the HTTP proxy integration or the HTTP custom integration.

API Gateway supports the following endpoint ports: 80, 443 and 1024-65535.

With proxy integration, setup is simple. You only need to set the HTTP method and the HTTP endpoint URI, according to the backend requirements, if you are not concerned with content encoding or caching.

With custom integration, setup is more involved. In addition to the proxy integration setup steps, you need to specify how the incoming request data is mapped to the integration request and how the resulting integration response data is mapped to the method response.

Topics

- Set up HTTP proxy integrations in API Gateway
- Set up HTTP custom integrations in API Gateway

Set up HTTP proxy integrations in API Gateway

To set up a proxy resource with the HTTP proxy integration type, create an API resource with a greedy path parameter (for example, `/parent/{proxy+}`) and integrate this resource with an HTTP backend endpoint (for example, `https://petstore-demo-endpoint.execute-api.com/petstore/{proxy}`) on the `ANY` method. The greedy path parameter must be at the end of the resource path.

As with a non-proxy resource, you can set up a proxy resource with the HTTP proxy integration by using the API Gateway console, importing an OpenAPI definition file, or calling the API Gateway REST API directly. For detailed instructions about using the API Gateway console to configure a proxy resource with the HTTP integration, see Tutorial: Build a REST API with HTTP proxy integration.

The following OpenAPI definition file shows an example of an API with a proxy resource that is integrated with the PetStore website.

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-09-12T23:19:28Z",
      "title": "PetStoreWithProxyResource"
   },
   "paths": {
      "/{proxy+}": {
         "x-amazon-apigateway-any-method": {
            "parameters": [
               {
                  "name": "proxy",
                  "in": "path",
                  "required": true,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {},
            "x-amazon-apigateway-integration": {
               "responses": {
                  "default": {
                     "statusCode": "200"
                  }
               },
               "requestParameters": {
                  "integration.request.path.proxy": "method.request.path.proxy"
               },
               "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/
{proxy}",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "ANY",
               "cacheNamespace": "rbftud",
               "cacheKeyParameters": [
                  "method.request.path.proxy"
               ],
               "type": "http_proxy"
            }
         }
      }
   },
   "servers": [
      {
         "url": "https://4z9giyi2c1.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/test"
            }
         }
      }
   ]
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-12T23:19:28Z",
    "title": "PetStoreWithProxyResource"
  },
  "host": "4z9giyi2c1.execute-api.us-east-1.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.path.proxy": "method.request.path.proxy"
          },
          "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/{proxy}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "ANY",
          "cacheNamespace": "rbftud",
          "cacheKeyParameters": [
            "method.request.path.proxy"
          ],
          "type": "http_proxy"
        }
      }
    }
  }
}
```

In this example, a cache key is declared on the `method.request.path.proxy` path parameter of the proxy resource. This is the default setting when you create the API using the API Gateway console. The API's base path (`/test`, corresponding to a stage) is mapped to the website's PetStore page (`/petstore`). The single integration request mirrors the entire PetStore website using the API's greedy path variable and the catch-all ANY method. The following examples illustrate this mirroring.

- **Set ANY as GET and `{proxy+}` as `pets`**

  Method request initiated from the frontend:

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets HTTP/1.1
  ```

  Integration request sent to the backend:

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets HTTP/1.1
  ```

The run-time instances of the ANY method and proxy resource are both valid. The call returns a `200 OK` response with the payload containing the first batch of pets, as returned from the backend.

- **Set ANY as GET and `{proxy+}` as `pets?type=dog`**

```
GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets?type=dog
 HTTP/1.1
```

Integration request sent to the backend:

```
GET http://petstore-demo-endpoint.execute-api.com/petstore/pets?type=dog HTTP/1.1
```

The run-time instances of the ANY method and proxy resource are both valid. The call returns a `200 OK` response with the payload containing the first batch of specified dogs, as returned from the backend.

- **Set ANY as GET and `{proxy+}` as `pets/{petId}`**

Method request initiated from the frontend:

```
GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets/1 HTTP/1.1
```

Integration request sent to the backend:

```
GET http://petstore-demo-endpoint.execute-api.com/petstore/pets/1 HTTP/1.1
```

The run-time instances of the ANY method and proxy resource are both valid. The call returns a `200 OK` response with the payload containing the specified pet, as returned from the backend.

- **Set ANY as POST and `{proxy+}` as `pets`**

Method request initiated from the frontend:

```
POST https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets HTTP/1.1
Content-Type: application/json
Content-Length: ...

{
  "type" : "dog",
  "price" : 1001.00
}
```

Integration request sent to the backend:

```
POST http://petstore-demo-endpoint.execute-api.com/petstore/pets HTTP/1.1
Content-Type: application/json
Content-Length: ...

{
  "type" : "dog",
  "price" : 1001.00
}
```

The run-time instances of the `ANY` method and proxy resource are both valid. The call returns a `200 OK` response with the payload containing the newly created pet, as returned from the backend.

- **Set ANY as `GET` and `{proxy+}` as `pets/cat`**

  Method request initiated from the frontend:

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets/cat
  ```

  Integration request sent to the backend:

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets/cat
  ```

  The run-time instance of the proxy resource path does not correspond to a backend endpoint and the resulting request is invalid. As a result, a `400 Bad Request` response is returned with the following error message.

  ```
  {
    "errors": [
      {
        "key": "Pet2.type",
        "message": "Missing required field"
      },
      {
        "key": "Pet2.price",
  ```

```
      "message": "Missing required field"
    }
  ]
}
```

- **Set ANY as GET and `{proxy+}` as `null`**

  Method request initiated from the frontend:

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test
  ```

  Integration request sent to the backend:

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets
  ```

  The targeted resource is the parent of the proxy resource, but the run-time instance of the `ANY` method is not defined in the API on that resource. As a result, this `GET` request returns a `403 Forbidden` response with the `Missing Authentication Token` error message as returned by API Gateway. If the API exposes the `ANY` or `GET` method on the parent resource (/), the call returns a `404 Not Found` response with the `Cannot GET /petstore` message as returned from the backend.

For any client request, if the targeted endpoint URL is invalid or the HTTP verb is valid but not supported, the backend returns a `404 Not Found` response. For an unsupported HTTP method, a `403 Forbidden` response is returned.

Set up HTTP custom integrations in API Gateway

With the HTTP custom integration, you have more control of which data to pass between an API method and an API integration and how to pass the data. You do this using data mappings.

As part of the method request setup, you set the requestParameters property on a Method resource. This declares which method request parameters, which are provisioned from the client, are to be mapped to integration request parameters or applicable body properties before being dispatched to the backend. Then, as part of the integration request setup, you set the requestParameters property on the corresponding Integration resource to specify the parameter-to-parameter mappings. You also set the requestTemplates property to specify mapping templates, one for each supported content type. The mapping templates map method request parameters, or body, to the integration request body.

Similarly, as part of the method response setup, you set the responseParameters property on the MethodResponse resource. This declares which method response parameters, to be dispatched to the client, are to be mapped from integration response parameters or certain applicable body properties that were returned from the backend. Then, as part of the integration response setup, you set the responseParameters property on the corresponding IntegrationResponse resource to specify the parameter-to-parameter mappings. You also set the responseTemplates map to specify mapping templates, one for each supported content type. The mapping templates map integration response parameters, or integration response body properties, to the method response body.

For more information about setting up mapping templates, see Setting up data transformations for REST APIs.

Set up API Gateway private integrations

The API Gateway private integration makes it simple to expose your HTTP/HTTPS resources within an Amazon VPC for access by clients outside of the VPC. To extend access to your private VPC resources beyond the VPC boundaries, you can create an API with private integration. You can control access to your API by using any of the authorization methods that API Gateway supports.

To create a private integration, you must first create a Network Load Balancer. Your Network Load Balancer must have a listener that routes requests to resources in your VPC. To improve the availability of your API, ensure that your Network Load Balancer routes traffic to resources in more than one Availability Zone in the AWS Region. Then, you create a VPC link that you use to connect your API and your Network Load Balancer. After you create a VPC link, you create private integrations to route traffic from your API to resources in your VPC through your VPC link and Network Load Balancer.

Note

The Network Load Balancer and API must be owned by the same AWS account.

With the API Gateway private integration, you can enable access to HTTP/HTTPS resources within a VPC without detailed knowledge of private network configurations or technology-specific appliances.

Topics

- Grant permissions to create a VPC link
- Set up an API Gateway API with private integrations using the API Gateway console
- Set up an API Gateway API with private integrations using the AWS CLI
- Set up API with private integrations using OpenAPI
- API Gateway accounts used for private integrations

Set up a Network Load Balancer for API Gateway private integrations

The following procedure outlines the steps to set up a Network Load Balancer (NLB) for API Gateway private integrations using the Amazon EC2 console and provides references for detailed instructions for each step.

For each VPC you have resources in, you only need to configure one NLB and one VPCLink. The NLB supports multiple listeners and target groups per NLB. You can configure each service as a specific listener on the NLB and use a single VPCLink to connect to the NLB. When creating the private integration in API Gateway you then define each service using the specific port that is assigned for each service. For more information, see the section called “Tutorial: Build an API with private integration”.

> **Note**
>
> The Network Load Balancer and API must be owned by the same AWS account.

To create a Network Load Balancer for private integration using the API Gateway console

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Set up a web server on an Amazon EC2 instance. For an example setup, see Installing a LAMP Web Server on Amazon Linux 2.
3. Create a Network Load Balancer, register the EC2 instance with a target group, and add the target group to a listener of the Network Load Balancer. For details, follow the instructions in Getting Started with Network Load Balancers.
4. After the Network Load Balancer is created, do the following:

    a. Note the ARN of the Network Load Balancer. You will need it to create a VPC link in API Gateway for integrating the API with the VPC resources behind the Network Load Balancer

b. Turn off security group evaluation for PrivateLink. Use the following command to turn off inbound rules on PrivateLink traffic.

```
aws elbv2 set-security-groups --load-balancer-
arn arn:aws:elasticloadbalancing:us-east-2:111122223333:loadbalancer/net/my-
loadbalancer/abc12345 \
  --security-groups sg-123345a --enforce-security-group-inbound-rules-on-
private-link-traffic off
```

Note

Do not add any dependencies to API Gateway CIDRs as they are bound to change without notice.

Grant permissions to create a VPC link

For you or a user in your account to create and maintain a VPC link, you or the user must have permissions to create, delete, and view VPC endpoint service configurations, change VPC endpoint service permissions, and examine load balancers. To grant such permissions, use the following steps.

To grant permissions to create, update, and delete a VPC link

1. Create an IAM policy similar to the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST",
                "apigateway:GET",
                "apigateway:PATCH",
                "apigateway:DELETE"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/vpclinks",
                "arn:aws:apigateway:us-east-1::/vpclinks/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "elasticloadbalancing:DescribeLoadBalancers"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateVpcEndpointServiceConfiguration",
                "ec2:DeleteVpcEndpointServiceConfigurations",
                "ec2:DescribeVpcEndpointServiceConfigurations",
                "ec2:ModifyVpcEndpointServicePermissions"
            ],
            "Resource": "*"
        }
    ]
}
```

2. Create or choose an IAM role and attach the preceding policy to the role.
3. Assign the IAM role to you or a user in your account who is creating VPC links.

Set up an API Gateway API with private integrations using the API Gateway console

For instructions using the API Gateway Console to set up an API with private integration, see Tutorial: Build a REST API with API Gateway private integration.

Set up an API Gateway API with private integrations using the AWS CLI

Before creating an API with the private integration, you must have your VPC resource set up and a Network Load Balancer created and configured with your VPC source as the target. If the requirements are not met, follow Set up a Network Load Balancer for API Gateway private integrations to install the VPC resource, create a Network Load Balancer, and set the VPC resource as a target of the Network Load Balancer.

> **Note**
>
> The Network Load Balancer and API must be owned by the same AWS account.

For you to be able to create and manage a VpcLink, you must also have the appropriate permissions configured. For more information, see Grant permissions to create a VPC link.

> **Note**
>
> You only need the permissions to create a VpcLink in your API. You do not need the permissions to use the VpcLink.

After the Network Load Balancer is created, note its ARN. You need it to create a VPC link for the private integration.

To set up an API with the private integration using AWS CLI

1. Create a VpcLink targeting the specified Network Load Balancer.

```
aws apigateway create-vpc-link \
    --name my-test-vpc-link \
    --target-arns arn:aws:elasticloadbalancing:us-east-2:123456789012:loadbalancer/
net/my-vpclink-test-nlb/1234567890abcdef
```

   The output of this command acknowledges the receipt of the request and shows the PENDING status for the VpcLink being created.

```
{
    "status": "PENDING",
    "targetArns": [
        "arn:aws:elasticloadbalancing:us-east-2:123456789012:loadbalancer/net/my-
vpclink-test-nlb/1234567890abcdef"
    ],
    "id": "gim7c3",
    "name": "my-test-vpc-link"
}
```

   It takes 2-4 minutes for API Gateway to finish creating the VpcLink. When the operation finishes successfully, the status is AVAILABLE. You can verify this by calling the following CLI command:

```
aws apigateway get-vpc-link --vpc-link-id gim7c3
```

If the operation fails, you get a `FAILED` status, with the `statusMessage` containing the error message. For example, if you attempt to create a `VpcLink` with a Network Load Balancer that is already associated with a VPC endpoint, you get the following on the `statusMessage` property:

```
"NLB is already associated with another VPC Endpoint Service"
```

After the `VpcLink` is created successfully, you can create an API and integrate it with the VPC resource through the `VpcLink`.

Note the `id` value of the newly created `VpcLink` (*`gim7c3`* in the preceding output). You need it to set up the private integration.

2. Set up an API by creating an API Gateway `RestApi` resource:

   ```
   aws apigateway create-rest-api --name 'My VPC Link Test'
   ```

   Note the `RestApi`'s `id` value in the returned result. You need this value to perform further operations on the API.

   For illustration purposes, we will create an API with only a `GET` method on the root resource (`/`) and integrate the method with the `VpcLink`.

3. Set up the `GET /` method. First get the identifier of the root resource (`/`):

   ```
   aws apigateway get-resources --rest-api-id  abcdef123
   ```

   In the output, note the `id` value of the `/` path. In this example, we assume it to be *`skpp60rab7`*.

   Set up the method request for the API method of `GET /`:

   ```
   aws apigateway put-method \
          --rest-api-id  abcdef123 \
          --resource-id skpp60rab7 \
          --http-method GET \
          --authorization-type "NONE"
   ```

If you do not use the proxy integration with the `VpcLink`, you must also set up at least a method response of the `200` status code. We will use the proxy integration here.

4. Set up the private integration of the `HTTP_PROXY` type and call the `put-integration` command as follows:

```
aws apigateway put-integration \
    --rest-api-id abcdef123 \
    --resource-id skpp60rab7 \
    --uri 'http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com' \
    --http-method GET \
    --type HTTP_PROXY \
    --integration-http-method GET \
    --connection-type VPC_LINK \
    --connection-id gim7c3
```

For a private integration, set `connection-type` to `VPC_LINK` and set `connection-id` to either your `VpcLink`'s identifier or a stage variable referencing your `VpcLink` ID. The `uri` parameter is not used for routing requests to your endpoint, but is used for setting the `Host` header and for certificate validation.

The command returns the following output:

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "connectionId": "gim7c3",
    "uri": "http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com",
    "connectionType": "VPC_LINK",
    "httpMethod": "GET",
    "cacheNamespace": "skpp60rab7",
    "type": "HTTP_PROXY",
    "cacheKeyParameters": []
}
```

Using a stage variable, you set the `connectionId` property when creating the integration:

```
aws apigateway put-integration \
    --rest-api-id abcdef123 \
    --resource-id skpp60rab7 \
    --uri 'http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com' \
    --http-method GET \
    --type HTTP_PROXY \
    --integration-http-method GET \
    --connection-type VPC_LINK \
    --connection-id "\${stageVariables.vpcLinkId}"
```

Make sure to double-quote the stage variable expression (`${stageVariables.vpcLinkId}`) and escape the `$` character.

Alternatively, you can update the integration to reset the `connectionId` value with a stage variable:

```
 aws apigateway update-integration \
    --rest-api-id abcdef123 \
    --resource-id skpp60rab7 \
    --http-method GET \
    --patch-operations '[{"op":"replace","path":"/
connectionId","value":"${stageVariables.vpcLinkId}"}]'
```

Make sure to use a stringified JSON list as the `patch-operations` parameter value.

You can use a stage variable to integrate your API with a different VPC or Network Load Balancer by resetting the `VpcLinks` stage variable value.

Because we used the private proxy integration, the API is now ready for deployment and for test runs. With the non-proxy integration, you must also set up the method response and integration response, just as you would when setting up an API with HTTP custom integrations.

5. To test the API, deploy the API. This is necessary if you have used the stage variable as a placeholder of the `VpcLink` ID. To deploy the API with a stage variable, call the `create-deployment` command as follows:

```
aws apigateway create-deployment \
    --rest-api-id abcdef123 \
    --stage-name test \
    --variables vpcLinkId=gim7c3
```

To update the stage variable with a different `VpcLink` ID (e.g., *`asf9d7`*), call the `update-stage` command:

```
aws apigateway update-stage \
    --rest-api-id abcdef123 \
    --stage-name test \
    --patch-operations op=replace,path='/variables/vpcLinkId',value='asf9d7'
```

Use the following command to invoke your API:

```
curl -X GET https://abcdef123.execute-api.us-east-2.amazonaws.com/test
```

Alternatively, you can type the API's invoke-URL in a web browser to view the result.

When you hardcode the `connection-id` property with the `VpcLink` ID literal, you can also call `test-invoke-method` to test invoking the API before it is deployed.

Set up API with private integrations using OpenAPI

You can set up an API with the private integration by importing the API's OpenAPI file. The settings are similar to the OpenAPI definitions of an API with HTTP integrations, with the following exceptions:

- You must explicitly set `connectionType` to `VPC_LINK`.
- You must explicitly set `connectionId` to the ID of a `VpcLink` or to a stage variable referencing the ID of a `VpcLink`.
- The `uri` parameter in the private integration points to an HTTP/HTTPS endpoint in the VPC, but is used instead to set up the integration request's `Host` header.
- The `uri` parameter in the private integration with an HTTPS endpoint in the VPC is used to verify the stated domain name against the one in the certificate installed on the VPC endpoint.

You can use a stage variable to reference the `VpcLink` ID. Or you can assign the ID value directly to `connectionId`.

The following JSON-formatted OpenAPI file shows an example of an API with a VPC link as referenced by a stage variable (`${stageVariables.vpcLinkId}`):

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2017-11-17T04:40:23Z",
    "title": "MyApiWithVpcLink"
  },
  "host": "p3wocvip9a.execute-api.us-west-2.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "uri": "http://my-vpclink-test-nlb-1234567890abcdef.us-
east-2.amazonaws.com",
          "passthroughBehavior": "when_no_match",
          "connectionType": "VPC_LINK",
          "connectionId": "${stageVariables.vpcLinkId}",
          "httpMethod": "GET",
          "type": "http_proxy"
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

API Gateway accounts used for private integrations

The following region-specific API Gateway account IDs are automatically added to your VPC endpoint service as `AllowedPrincipals` when you create a `VpcLink`.

Region	Account ID
us-east-1	392220576650
us-east-2	718770453195
us-west-1	968246515281
us-west-2	109351309407
ca-central-1	796887884028
eu-west-1	631144002099
eu-west-2	544388816663
eu-west-3	061510835048
eu-central-1	474240146802
eu-central-2	166639821150
eu-north-1	394634713161
eu-south-1	753362059629
eu-south-2	359345898052
ap-northeast-1	969236854626
ap-northeast-2	020402002396

Region	Account ID
ap-northeast-3	360671645888
ap-southeast-1	195145609632
ap-southeast-2	798376113853
ap-southeast-3	652364314486
ap-southeast-4	849137399833
ap-south-1	507069717855
ap-south-2	644042651268
ap-east-1	174803364771
sa-east-1	287228555773
me-south-1	855739686837
me-central-1	614065512851

Set up mock integrations in API Gateway

Amazon API Gateway supports mock integrations for API methods. This feature enables API developers to generate API responses from API Gateway directly, without the need for an integration backend. As an API developer, you can use this feature to unblock dependent teams that need to work with an API before the project development is complete. You can also use this feature to provision a landing page for your API, which can provide an overview of and navigation to your API. For an example of such a landing page, see the integration request and response of the GET method on the root resource of the example API discussed in Tutorial: Create a REST API by importing an example.

As an API developer, you decide how API Gateway responds to a mock integration request. For this, you configure the method's integration request and integration response to associate a response with a given status code. For a method with the mock integration to return a `200` response, configure the integration request body mapping template to return the following.

```
{"statusCode": 200}
```

Configure a 200 integration response to have the following body mapping template, for example:

```
{
    "statusCode": 200,
    "message": "Go ahead without me."
}
```

Similarly, for the method to return, for example, a 500 error response, set up the integration request body mapping template to return the following.

```
{"statusCode": 500}
```

Set up a 500 integration response with, for example, the following mapping template:

```
{
    "statusCode": 500,
    "message": "The invoked method is not supported on the API resource."
}
```

Alternatively, you can have a method of the mock integration return the default integration response without defining the integration request mapping template. The default integration response is the one with an undefined **HTTP status regex**. Make sure appropriate passthrough behaviors are set.

> **Note**
>
> Mock integrations aren't intended to support large response templates. If you need them for your use case, you should consider using a Lambda integration instead.

Using an integration request mapping template, you can inject application logic to decide which mock integration response to return based on certain conditions. For example, you could use a scope query parameter on the incoming request to determine whether to return a successful response or an error response:

```
{
  #if( $input.params('scope') == "internal" )
    "statusCode": 200
  #else
    "statusCode": 500
  #end
}
```

This way, the method of the mock integration lets internal calls to go through while rejecting other types of calls with an error response.

In this section, we describe how to use the API Gateway console to enable the mock integration for an API method.

Topics

- Enable mock integration using the API Gateway console

Enable mock integration using the API Gateway console

You must have a method available in API Gateway. Follow the instructions in Tutorial: Build a REST API with HTTP non-proxy integration.

1. Choose an API resource and choose **Create method**.

   To create the method, do the following:

   a. For **Method type**, select a method.
   b. For **Integration type**, select **Mock**.
   c. Choose **Create method**.
   d. On the **Method request** tab, for **Method request settings**, choose **Edit**.
   e. Choose **URL query string parameters**. Choose **Add query string** and for **Name**, enter **`scope`**. This query parameter determines if the caller is internal or otherwise.
   f. Choose **Save**.
2. On the **Method response** tab, choose **Create response**, and then do the following:

   a. For **HTTP Status**, enter **`500`**.
   b. Choose **Save**.
3. On the **Integration request** tab, for **Integration request settings**, choose **Edit**.

4. Choose **Mapping templates**, and then do the following:

   a. Choose **Add mapping template**.

   b. For **Content type**, enter **`application/json`**.

   c. For **Template body**, enter the following:

   ```
   {
     #if( $input.params('scope') == "internal" )
       "statusCode": 200
     #else
       "statusCode": 500
     #end
   }
   ```

   d. Choose **Save**.

5. On the **Integration response** tab, for the **Default - Response** choose **Edit**.

6. Choose **Mapping templates**, and then do the following:

   a. For **Content type**, enter **`application/json`**.

   b. For **Template body**, enter the following:

   ```
   {
       "statusCode": 200,
       "message": "Go ahead without me"
   }
   ```

   c. Choose **Save**.

7. Choose **Create response**.

   To create a 500 response, do the following:

   a. For **HTTP status regex**, enter **`5\d{2}`**.

   b. For **Method response status**, select **`500`**.

   c. Choose **Save**.

   d. For **5\d{2} - Response**, choose **Edit**.

   e. Choose **Mapping templates**, and then choose **Add mapping template**.

   f. For **Content type**, enter **`application/json`**.

   g. For **Template body**, enter the following:

```
{
    "statusCode": 500,
    "message": "The invoked method is not supported on the API resource."
}
```

h. Choose **Save**.

8. Choose the **Test** tab. You might need to choose the right arrow button to show the tab. To test your mock integration, do the following:

   a. Enter `scope=internal` under **Query strings**. Choose **Test**. The test result shows:

```
Request: /?scope=internal
Status: 200
Latency: 26 ms
Response Body

{
  "statusCode": 200,
  "message": "Go ahead without me"
}

Response Headers

{"Content-Type":"application/json"}
```

   b. Enter `scope=public` under `Query strings` or leave it blank. Choose **Test**. The test result shows:

```
Request: /
Status: 500
Latency: 16 ms
Response Body

{
  "statusCode": 500,
  "message": "The invoked method is not supported on the API resource."
}
```

```
Response Headers

{"Content-Type":"application/json"}
```

You can also return headers in a mock integration response by first adding a header to the method response and then setting up a header mapping in the integration response. In fact, this is how the API Gateway console enables CORS support by returning CORS required headers.

Use request validation in API Gateway

You can configure API Gateway to perform basic validation of an API request before proceeding with the integration request. When the validation fails, API Gateway immediately fails the request, returns a 400 error response to the caller, and publishes the validation results in CloudWatch Logs. This reduces unnecessary calls to the backend. More importantly, it lets you focus on the validation efforts specific to your application. You can validate a request body by verifying that required request parameters are valid and non-null or by specifying a model schema for more complicated data validation.

Topics

Overview of basic request validation in API Gateway

API Gateway can perform the basic request validation, so that you can focus on app-specific validation in the backend. For validation, API Gateway verifies either or both of the following conditions:

- The required request parameters in the URI, query string, and headers of an incoming request are included and not blank.
- The applicable request payload adheres to the configured JSON schema request of the method.

To turn on validation, you specify validation rules in a request validator, add the validator to the API's map of request validators, and assign the validator to individual API methods.

Note

Request body validation and Integration passthrough behaviors are two separate topics. When a request payload does not have a matching model schema, you can choose to passthrough or block the original payload. For more information, see Integration passthrough behaviors.

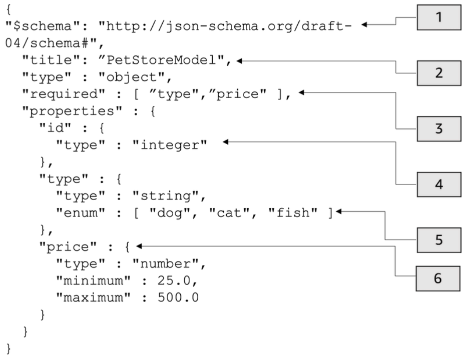
Understanding data models

In API Gateway, a model defines the data structure of a payload. In API Gateway, models are defined using the JSON schema draft 4. The following JSON object is sample data in the Pet Store example.

```
{
    "id": 1,
    "type": "dog",
    "price": 249.99
}
```

The data contains the `id`, `type`, and `price` of the pet. A model of this data allows you to:

- Use basic request validation.
- Create mapping templates for data transformation.
- Create a user-defined data type (UDT) when you generate an SDK.



```
{
"$schema": "http://json-schema.org/draft-
04/schema#",
  "title": ”PetStoreModel",
  "type" : "object",
  "required" : [ ”type",”price" ],
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
      "type" : "number",
      "minimum" : 25.0,
      "maximum" : 500.0
    }
  }
}
```

In this model:

1. The `$schema` object represents a valid JSON Schema version identifier. This schema is the JSON Schema draft v4.
2. The `title` object is a human-readable identifier for the model. This title is `PetStoreModel`.
3. The `required` validation keyword requires `type`, and `price` for basic request validation.
4. The `properties` of the model are `id`, `type`, and `price`. Each object has properties that are described in the model.
5. The object `type` can have only the values `dog`, `cat`, or `fish`.
6. The object `price` is a number and is constrained with a `minimum` of 25 and a `maximum` of 500.

PetStore model

```
1 {
2 "$schema": "http://json-schema.org/draft-04/schema#",
3  "title": "PetStoreModel",
4  "type" : "object",
5  "required" : [ "price", "type" ],
6  "properties" : {
7    "id" : {
8      "type" : "integer"
9    },
10    "type" : {
11      "type" : "string",
12      "enum" : [ "dog", "cat", "fish" ]
13    },
14    "price" : {
15      "type" : "number",
16      "minimum" : 25.0,
17      "maximum" : 500.0
18    }
19  }
20 }
```

In this model:

1. On line 2, the `$schema` object represents a valid JSON Schema version identifier. This schema is the JSON Schema draft v4.

2. On line 3, the `title` object is a human-readable identifier for the model. This title is `PetStoreModel`.
3. On line 5, the `required` validation keyword requires `type`, and `price` for basic request validation.
4. On lines 6 -- 17, the `properties` of the model are `id`, `type`, and `price`. Each object has properties that are described in the model.
5. On line 12, the object `type` can have only the values `dog`, `cat`, or `fish`.
6. On lines 14 -- 17, the object `price` is a number and is constrained with a `minimum` of 25 and a `maximum` of 500.

Creating more complex models

You can use the `$ref` primitive to create reusable definitions for longer models. For example, you can create a definition called `Price` in the `definitions` section describing the `price` object. The value of `$ref` is the `Price` definition.

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStoreModelReUsableRef",
  "required" : ["price", "type" ],
  "type" : "object",
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
        "$ref": "#/definitions/Price"
    }
  },
  "definitions" : {
      "Price": {
        "type" : "number",
        "minimum" : 25.0,
        "maximum" : 500.0
            }
      }
}
```

You can also reference another model schema defined in an external model file. Set the value of the `$ref` property to the location of the model. In the following example, the `Price` model is defined in the `PetStorePrice` model in API `a1234`.

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStorePrice",
  "type": "number",
  "minimum": 25,
  "maximum": 500
}
```

The longer model can reference the `PetStorePrice` model.

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStoreModelReusableRefAPI",
  "required" : [ "price", "type" ],
  "type" : "object",
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
        "$ref": "https://apigateway.amazonaws.com/restapis/a1234/models/PetStorePrice"
    }
  }
}
```

Using output data models

If you transform your data, you can define a payload model in the integration response. A payload model can be used when you generate an SDK. For strongly typed languages, such as Java, Objective-C, or Swift, the object corresponds to a user-defined data type (UDT). API Gateway creates a UDT if you provide it with a data model when you generate an SDK. For more information about data transformations, see Understanding mapping templates.

Output data

```
{
                [
  {
    "description" : "Item 1 is a
 dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a
 cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a
 fish.",
    "askingPrice" : 0.99
  }
]
}
```

Output model

```
{
"$schema": "http://json-schema.org/
draft-04/schema#",
  "title": ”PetStoreOutputModel",
  "type" : "object",
  "required" : [ "description",
 "askingPrice" ],
  "properties" : {
    "description" : {
      "type" : "string"
    },
    "askingPrice" : {
      "type" : "number",
      "minimum" : 25.0,
      "maximum" : 500.0
    }
  }
}
```

With this model, you can call an SDK to retrieve the `description` and `askingPrice` property values by reading the `PetStoreOutputModel[i].description` and `PetStoreOutputModel[i].askingPrice` properties. If no model is provided, API Gateway uses the empty model to create a default UDT.

Next steps

- This section provides resources that you can use to gain more knowledge about the concepts presented in this topic.

  You can follow the request validation tutorials:
  - Set up request validation using the API Gateway console
  - Set up basic request validation using the AWS CLI
  - Set up basic request validation using an OpenAPI definition
- You can get more information about data transformation and mapping templates, Understanding mapping templates.
- You can also see a more advanced photo album example model. See Photos example.

Set up basic request validation in API Gateway

This section shows how to set up request validation for API Gateway using the console, AWS CLI, and an OpenAPI definition.

Topics

- Set up request validation using the API Gateway console
- Set up basic request validation using the AWS CLI
- Set up basic request validation using an OpenAPI definition

Set up request validation using the API Gateway console

You can use the API Gateway console to validate a request by selecting one of three validators for an API request:

- **Validate body**.
- **Validate query string parameters and headers**.
- **Validate body, query string parameters, and headers**.

When you apply one of the validators on an API method, the API Gateway console adds the validator to the API's RequestValidators map.

To follow this tutorial, you'll use an AWS CloudFormation template to create an incomplete API Gateway API. This API has a `/validator` resource with `GET` and `POST` methods. Both methods are integrated with the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP endpoint. You will configure two kinds of request validation:

- In the `GET` method, you will configure request validation for URL query string parameters.
- In the `POST` method, you will configure request validation for the request body.

This will allow only certain API calls to pass through to the API.

Download and unzip the app creation template for AWS CloudFormation. You'll use this template to create an incomplete API. You will finish the rest of the steps in the API Gateway console.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.
3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`request-validation-tutorial-console`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, you're ready to move on to the next step.

To select your newly created API

1. Select the newly created **`request-validation-tutorial-console`** stack.

2. Choose **Resources**.
3. Under **Physical ID**, choose your API. This link will direct you to the API Gateway console.

Before you modify the `GET` and `POST` methods, you must create a model.

To create a model

1. A model is required to use request validation on the body of an incoming request. To create a model, in the main navigation pane, choose **Models**.
2. Choose **Create model**.
3. For **Name**, enter **`PetStoreModel`**.
4. For **Content Type**, enter **`application/json`**. If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **`$default`**.
5. For **Description**, enter **`My PetStore Model`** as the model description.
6. For **Model schema**, paste the following model into the code editor, and choose **Create**.

```
{
  "type" : "object",
  "required" : [ "name", "price", "type" ],
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "name" : {
      "type" : "string"
    },
    "price" : {
      "type" : "number",
      "minimum" : 25.0,
      "maximum" : 500.0
    }
  }
}
```

For more information about the model, see Understanding data models.

To configure request validation for a GET method

1. In the main navigation pane, choose **Resources**, and then select the **GET** method.
2. On the **Method request** tab, under **Method request settings**, choose **Edit**.
3. For **Request validator**, select **Validate query string parameters and headers**.
4. Under **URL query string parameters**, do the following:

    a. Choose **Add query string**.

    b. For **Name**, enter **`petType`**.

    c. Turn on **Required**.

    d. Keep **Caching** turned off.
5. Choose **Save**.
6. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
7. Under **URL query string parameters**, do the following:

    a. Choose **Add query string**.

    b. For **Name**, enter **`petType`**.

    c. For **Mapped from**, enter **`method.request.querystring.petType`**. This maps the **`petType`** to the pet's type.

    For more information about data mapping, see the data mapping tutorial.

    d. Keep **Caching** turned off.
8. Choose **Save**.

To test request validation for the GET method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Query strings**, enter **`petType=dog`**, and then choose **Test**.
3. The method test will return `200 OK` and a list of dogs.

    For information about how to transform this output data, see the data mapping tutorial.
4. Remove **`petType=dog`** and choose **Test**.
5. The method test will return a `400` error with the following error message:

```
{
  "message": "Missing required request parameters: [petType]"
}
```

To configure request validation for the POST method

1. In the main navigation pane, choose **Resources**, and then select the **POST** method.
2. On the **Method request** tab, under **Method request settings**, choose **Edit**.
3. For **Request validator**, select **Validate body**.
4. Under **Request body**, choose **Add model**.
5. For **Content type**, enter **`application/json`**, and then for **Model**, select **PetStoreModel**.
6. Choose **Save**.

To test request validation for a POST method

1. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. For **Request body** paste the following into the code editor:

   ```
   {
     "id": 2,
     "name": "Bella",
     "type": "dog",
     "price": 400
   }
   ```

   Choose **Test**.
3. The method test will return `200 OK` and a success message.
4. For **Request body** paste the following into the code editor:

   ```
   {
     "id": 2,
     "name": "Bella",
     "type": "dog",
     "price": 4000
   }
   ```

Choose **Test**.

5. The method test will return a `400` error with the following error message:

```
{
  "message": "Invalid request body"
}
```

   At the bottom of the test logs, the reason for the invalid request body is returned. In this case, the price of the pet was outside the maximum specified in the model.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

Next steps

- For information about how to transform output data and perform more data mapping, see the data mapping tutorial.
- Follow the Set up basic request validation using the AWS CLI tutorial, to do similar steps using the AWS CLI.

Set up basic request validation using the AWS CLI

You can create a validator to set up request validation using the AWS CLI. To follow this tutorial, you'll use an AWS CloudFormation template to create an incomplete API Gateway API.

> **Note**
>
> This is not the same AWS CloudFormation template as the console tutorial.

Using a pre-exposed `/validator`resource, you will create `GET` and `POST` methods. Both methods will be integrated with the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP endpoint. You will configure the following two request validations:

- On the GET method, you will create a `params-only` validator to validate URL query string parameters.
- On the POST method, you will create a `body-only` validator to validate the request body.

This will allow only certain API calls to pass through to the API.

To create an AWS CloudFormation stack

Download and unzip the app creation template for AWS CloudFormation.

To complete the following tutorial, you need the AWS Command Line Interface (AWS CLI) version 2.

For long commands, an escape character (\) is used to split a command over multiple lines.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

1. Use the following command to create the AWS CloudFormation stack.

   ```
   aws cloudformation create-stack --stack-name request-validation-tutorial-cli
    --template-body file://request-validation-tutorial-cli.zip --capabilities
    CAPABILITY_NAMED_IAM
   ```

2. AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. Use the following command to see the status of your AWS CloudFormation stack.

   ```
   aws cloudformation describe-stacks --stack-name request-validation-tutorial-cli
   ```

3. When the status of your AWS CloudFormation stack is `StackStatus: "CREATE_COMPLETE"`, use the following command to retrieve relevant output values for future steps.

```
aws cloudformation describe-stacks --stack-name request-validation-tutorial-cli
--query "Stacks[*].Outputs[*].{OutputKey: OutputKey, OutputValue: OutputValue,
Description: Description}"
```

The output values are the following:

- ApiId, which is the ID for the API. For this tutorial, the API ID is `abc123`.
- ResourceId, which is the ID for the validator resource where the `GET` and `POST` methods are exposed. For this tutorial, the Resource ID is `efg456`

To create the request validators and import a model

1. A validator is required to use request validation with the AWS CLI. Use the following command to create a validator that validates only request parameters.

   ```
   aws apigateway create-request-validator --rest-api-id abc123 \
         --no-validate-request-body \
         --validate-request-parameters \
         --name params-only
   ```

   Note the ID of the `params-only` validator.

2. Use the following command to create a validator that validates only the request body.

   ```
   aws apigateway create-request-validator --rest-api-id abc123 \
         --validate-request-body \
         --no-validate-request-parameters \
         --name body-only
   ```

   Note the ID of the `body-only` validator.

3. A model is required to use request validation on the body of an incoming request. Use the following command to import a model.

   ```
   aws apigateway create-model --rest-api-id abc123 --name PetStoreModel --description
    'My PetStore Model' --content-type 'application/json' --schema '{"type":
    "object", "required" : [ "name", "price", "type" ], "properties" : { "id" :
    {"type" : "integer"},"type" : {"type" : "string", "enum" : [ "dog", "cat",
   ```

```
"fish" ]},"name" : { "type" : "string"},"price" : {"type" : "number","minimum" :
25.0, "maximum" : 500.0}}}}'
```

If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, specify `$default` as the key.

To create the GET and POST methods

1. Use the following command to add the `GET` HTTP method on the `/validate` resource. This command creates the `GET`method, adds the `params-only` validator, and sets the query string `petType` as required.

   ```
   aws apigateway put-method --rest-api-id abc123 \
          --resource-id efg456 \
          --http-method GET \
          --authorization-type "NONE" \
          --request-validator-id aaa111 \
          --request-parameters "method.request.querystring.petType=true"
   ```

   Use the following command to add the `POST` HTTP method on the `/validate` resource. This command creates the `POST`method, adds the `body-only` validator, and attaches the model to the body-only validator.

   ```
   aws apigateway put-method --rest-api-id abc123 \
          --resource-id efg456 \
          --http-method POST \
          --authorization-type "NONE" \
          --request-validator-id bbb222 \
          --request-models 'application/json'=PetStoreModel
   ```

2. Use the following command to set up the `200 OK` response of the `GET /validate` method.

   ```
   aws apigateway put-method-response --rest-api-id abc123  \
               --resource-id efg456 \
               --http-method GET \
               --status-code 200
   ```

   Use the following command to set up the `200 OK` response of the `POST /validate` method.

```
aws apigateway put-method-response --rest-api-id abc123  \
            --resource-id efg456 \
            --http-method POST \
            --status-code 200
```

3. Use the following command to set up an `Integration` with a specified HTTP endpoint for the `GET /validation` method.

```
aws apigateway put-integration --rest-api-id abc123  \
            --resource-id efg456 \
            --http-method GET \
            --type HTTP \
            --integration-http-method GET \
            --request-parameters '{"integration.request.querystring.type" :
 "method.request.querystring.petType"}' \
            --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets'
```

   Use the following command to set up an `Integration` with a specified HTTP endpoint for the `POST /validation` method.

```
aws apigateway put-integration --rest-api-id abc123  \
              --resource-id efg456 \
              --http-method POST \
              --type HTTP \
              --integration-http-method GET \
              --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets'
```

4. Use the following command to set up an integration response for the `GET /validation` method.

```
aws apigateway put-integration-response --rest-api-id abc123 \
              --resource-id efg456\
              --http-method GET \
              --status-code 200 \
              --selection-pattern ""
```

   Use the following command to set up an integration response for the `POST /validation` method.

```
aws apigateway put-integration-response --rest-api-id abc123 \
            --resource-id efg456 \
            --http-method POST \
            --status-code 200 \
            --selection-pattern ""
```

To test the API

1. To test the GET method, which will perform request validation for the query strings, use the following command:

   ```
   aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method GET \
               --path-with-query-string '/validate?petType=dog'
   ```

   The result will return a `200 OK` and list of dogs.

2. Use the following command to test without including the query string `petType`

   ```
   aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method GET
   ```

   The result will return a `400` error.

3. To test the POST method, which will perform request validation for the request body, use the following command:

   ```
    aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method POST \
               --body '{"id": 1, "name": "bella", "type": "dog", "price" : 400 }'
   ```

   The result will return a `200 OK` and a success message.

4. Use the following command to test using an invalid body.

   ```
    aws apigateway test-invoke-method --rest-api-id abc123 \
                 --resource-id efg456 \
                 --http-method POST \
                 --body '{"id": 1, "name": "bella", "type": "dog", "price" : 1000 }'
   ```

The result will return a `400` error, as the price of the dog is over the maximum price defined by the model.

To delete an AWS CloudFormation stack

- Use the following command to delete your AWS CloudFormation resources.

```
aws cloudformation delete-stack  --stack-name request-validation-tutorial-cli
```

Set up basic request validation using an OpenAPI definition

You can declare a request validator at the API level by specifying a set of the x-amazon-apigateway-request-validators.requestValidator object objects in the x-amazon-apigateway-request-validators object map to select what part of the request will be validated. In the example OpenAPI definition, there are two validators:

- `all` validator which validates both the body, using the `RequestBodyModel` data model, and the parameters.
- `param-only` which validates only the parameters.

To turn a request validator on all methods of an API, specify an x-amazon-apigateway-request-validator property property at the API level of the OpenAPI definition. In the example OpenAPI definition, the `all` validator is used on all API methods, unless otherwise overridden. When using a model to validate the body, if no matching content type is found, request validation is not performed. To use the same model regardless of the content type, specify `$default` as the key.

To turn on a request validator on an individual method, specify the `x-amazon-apigateway-request-validator` property at the method level. In the example, OpenAPI definition, the `param-only` validator overwrites the `all` validator on the `GET` method.

To import the OpenAPI example into API Gateway, see the following instructions to Import a regional API into API Gateway or to Import an edge-optimized API into API Gateway.

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "ReqValidators Sample",
    "version" : "1.0.0"
  },
  "servers" : [ {
    "url" : "/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "/v1"
      }
    }
  } ],
  "paths" : {
    "/validation" : {
      "get" : {
        "parameters" : [ {
          "name" : "q1",
          "in" : "query",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "headers" : {
              "test-method-response-header" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/ArrayOfError"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "params-only",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
 value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.querystring.type" : "method.request.querystring.q1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      },
      "post" : {
        "parameters" : [ {
          "name" : "h1",
          "in" : "header",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/RequestBodyModel"
              }
            }
          },
          "required" : true
        },
        "responses" : {
          "200" : {
            "description" : "200 response",
            "headers" : {
              "test-method-response-header" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/ArrayOfError"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "all",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "POST",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.custom_h1" : "method.request.header.h1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "RequestBodyModel" : {
        "required" : [ "name", "price", "type" ],
        "type" : "object",
        "properties" : {
          "id" : {
            "type" : "integer"
          },
          "type" : {
            "type" : "string",
            "enum" : [ "dog", "cat", "fish" ]
          },
          "name" : {
            "type" : "string"
          },
          "price" : {
            "maximum" : 500.0,
            "minimum" : 25.0,
            "type" : "number"
          }
        }
      },
      "ArrayOfError" : {
        "type" : "array",
        "items" : {
          "$ref" : "#/components/schemas/Error"
        }
      },
      "Error" : {
        "type" : "object"
      }
    }
  },
  "x-amazon-apigateway-request-validators" : {
    "all" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : true
    },
    "params-only" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : false
    }
  }
}
```

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "version" : "1.0.0",
    "title" : "ReqValidators Sample"
  },
  "basePath" : "/v1",
  "schemes" : [ "https" ],
  "paths" : {
    "/validation" : {
      "get" : {
        "produces" : [ "application/json", "application/xml" ],
        "parameters" : [ {
          "name" : "q1",
          "in" : "query",
          "required" : true,
          "type" : "string"
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "schema" : {
              "$ref" : "#/definitions/ArrayOfError"
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "params-only",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.querystring.type" : "method.request.querystring.q1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      },
      "post" : {
        "consumes" : [ "application/json" ],
        "produces" : [ "application/json", "application/xml" ],
        "parameters" : [ {
          "name" : "h1",
          "in" : "header",
          "required" : true,
          "type" : "string"
        }, {
          "in" : "body",
          "name" : "RequestBodyModel",
          "required" : true,
          "schema" : {
            "$ref" : "#/definitions/RequestBodyModel"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "schema" : {
              "$ref" : "#/definitions/ArrayOfError"
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "all",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "POST",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static 
 value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.custom_h1" : "method.request.header.h1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      }
    }
  },
  "definitions" : {
    "RequestBodyModel" : {
      "type" : "object",
      "required" : [ "name", "price", "type" ],
      "properties" : {
        "id" : {
          "type" : "integer"
        },
        "type" : {
          "type" : "string",
          "enum" : [ "dog", "cat", "fish" ]
        },
        "name" : {
          "type" : "string"
        },
        "price" : {
          "type" : "number",
          "minimum" : 25.0,
          "maximum" : 500.0
        }
      }
    },
    "ArrayOfError" : {
      "type" : "array",
      "items" : {
        "$ref" : "#/definitions/Error"
      }
    },
    "Error" : {
      "type" : "object"
    }
  },
  "x-amazon-apigateway-request-validators" : {
    "all" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : true
    },
    "params-only" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : false
    }
  }
}
```

OpenAPI definitions of a sample API with basic request validation

The following OpenAPI definition defines a sample API with request validation enabled. The API is a subset of the PetStore API. It exposes a POST method to add a pet to the pets collection and a GET method to query pets by a specified type.

There are two request validators declared in the x-amazon-apigateway-request-validators map at the API level. The params-only validator is enabled on the API and inherited by the GET method. This validator allows API Gateway to verify that the required query parameter (q1) is included and not blank in the incoming request. The all validator is enabled on the POST method. This validator verifies that the required header parameter (h1) is set and not blank. It also verifies that the payload format adheres to the specified RequestBodyModel If there is no matching content type is found, request validation is not performed. When using a model to validate the body, if no matching content type is found, request validation is not performed. To use the same model regardless of the content type, specify $default as the key.

This model requires that the input JSON object contains the name, type, and price properties. The name property can be any string, type must be one of the specified enumeration fields (["dog", "cat", "fish"]), and price must range between 25 and 500. The id parameter is not required.

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "title": "ReqValidators Sample",
    "version": "1.0.0"
  },
  "schemes": [
    "https"
  ],
  "basePath": "/v1",
  "produces": [
    "application/json"
  ],
  "x-amazon-apigateway-request-validators" : {
    "all" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  "x-amazon-apigateway-request-validator" : "params-only",
  "paths": {
    "/validation": {
      "post": {
        "x-amazon-apigateway-request-validator" : "all",
        "parameters": [
          {
            "in": "header",
            "name": "h1",
            "required": true
          },
          {
            "in": "body",
            "name": "RequestBodyModel",
            "required": true,
            "schema": {
              "$ref": "#/definitions/RequestBodyModel"
            }
          }
        ],
        "responses": {
          "200": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/Error"
              }
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "security" : [{
          "api_key" : []
        }],
        "x-amazon-apigateway-auth" : {
          "type" : "none"
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "httpMethod" : "POST",
          "requestParameters": {
            "integration.request.header.custom_h1": "method.request.header.h1"
          },
          "responses" : {
            "2\\d{2}" : {
              "statusCode" : "200"
            },
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
 value'"
              },
              "responseTemplates" : {
                "application/json" : "json 400 response template",
                "application/xml" : "xml 400 response template"
              }
            }
          }
        }
      },
      "get": {
        "parameters": [
          {
            "name": "q1",
            "in": "query",
            "required": true
          }
        ],
        "responses": {
          "200": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/Error"
              }
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "security" : [{
          "api_key" : []
        }],
        "x-amazon-apigateway-auth" : {
          "type" : "none"
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "httpMethod" : "GET",
          "requestParameters": {
            "integration.request.querystring.type": "method.request.querystring.q1"
          },
          "responses" : {
            "2\\d{2}" : {
              "statusCode" : "200"
            },
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static 
 value'"
              },
              "responseTemplates" : {
                "application/json" : "json 400 response template",
                "application/xml" : "xml 400 response template"
              }
            }
          }
        }
      }
    }
  },
  "definitions": {
    "RequestBodyModel": {
      "type": "object",
      "properties": {
        "id":    { "type": "integer" },
        "type":  { "type": "string", "enum": ["dog", "cat", "fish"] },
        "name":  { "type": "string" },
        "price": { "type": "number", "minimum": 25, "maximum": 500 }
      },
      "required": ["type", "name", "price"]
    },
    "Error": {
      "type": "object",
      "properties": {

      }
    }
  }
}
```

AWS CloudFormation template of a sample API with basic request validation

The following AWS CloudFormation example template definition defines a sample API with request validation enabled. The API is a subset of the PetStore API. It exposes a POST method to add a pet to the pets collection and a GET method to query pets by a specified type.

There are two request validators declared:

GETValidator

This validator is enabled on the GET method. It allows API Gateway to verify that the required query parameter (q1) is included and not blank in the incoming request.

POSTValidator

This validator is enabled on the POST method. It allows API Gateway to verify that payload request format adheres to the specified RequestBodyModel when the content type is application/json if no matching content type is found, request validation is not performed. To use the same model regardless of the content type, specify $default. RequestBodyModel contains an additional model, RequestBodyModelId, to define the pet ID.

```
AWSTemplateFormatVersion: 2010-09-09
Parameters:
  StageName:
    Type: String
    Default: v1
    Description: Name of API stage.
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: ReqValidatorsSample
  RequestBodyModelId:
    Type: 'AWS::ApiGateway::Model'
    Properties:
      RestApiId: !Ref Api
      ContentType: application/json
      Description: Request body model for Pet ID.
      Schema:
        $schema: 'http://json-schema.org/draft-04/schema#'
        title: RequestBodyModelId
        properties:
            id:
              type: integer
  RequestBodyModel:
    Type: 'AWS::ApiGateway::Model'
    Properties:
      RestApiId: !Ref Api
      ContentType: application/json
      Description: Request body model for Pet type, name, price, and ID.
      Schema:
        $schema: 'http://json-schema.org/draft-04/schema#'
        title: RequestBodyModel
        required:
          - price
          - name
          - type
        type: object
        properties:
            id:
              "$ref": !Sub
                - 'https://apigateway.amazonaws.com/restapis/${Api}/models/
${RequestBodyModelId}'
                - Api: !Ref Api
                  RequestBodyModelId: !Ref RequestBodyModelId
            price:
              type: number
              minimum: 25
              maximum: 500
            name:
              type: string
            type:
              type: string
              enum:
                - "dog"
                - "cat"
                - "fish"
  GETValidator:
    Type: AWS::ApiGateway::RequestValidator
    Properties:
      Name: params-only
      RestApiId: !Ref Api
      ValidateRequestBody: False
      ValidateRequestParameters: True
  POSTValidator:
    Type: AWS::ApiGateway::RequestValidator
    Properties:
      Name: body-only
      RestApiId: !Ref Api
      ValidateRequestBody: True
      ValidateRequestParameters: False
  ValidationResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'validation'
  ValidationMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref ValidationResource
      HttpMethod: GET
      AuthorizationType: NONE
      RequestValidatorId: !Ref GETValidator
      RequestParameters:
        method.request.querystring.q1: true
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  ValidationMethodPost:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref ValidationResource
      HttpMethod: POST
      AuthorizationType: NONE
      RequestValidatorId: !Ref POSTValidator
      RequestModels:
        application/json : !Ref RequestBodyModel
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: POST
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - ValidationMethodGet
      - RequestBodyModel
    Properties:
      RestApiId: !Ref Api
      StageName: !Sub '${StageName}'
Outputs:
  ApiRootUrl:
    Description: Root Url of the API
    Value: !Sub 'https://${Api}.execute-api.${AWS::Region}.amazonaws.com/${StageName}'
```

Setting up data transformations for REST APIs

In API Gateway, an API's method request can take a payload in a different format from the integration request payload. Similarly, the backend may return an integration response payload different from the method response payload. You can map URL path parameters, URL query string parameters, HTTP headers, and the request body across API Gateway using mapping templates.

A *mapping template* is a script expressed in Velocity Template Language (VTL) and applied to the payload using JSONPath expressions.

The payload can have a *data model* according to the JSON schema draft 4. To learn more about models, see Understanding data models.

> **Note**
>
> You don't have to define any model to create a mapping template, but you must define a model in order to have API Gateway to generate a SDK or to turn on request body validation for your API.

Topics

Understanding mapping templates

In API Gateway, an API's method request or response can take a payload in a different format from the integration request or response.

You can transform your data to:

- Match the payload to an API-specified format.
- Override an API's request and response parameters and status codes.
- Return client selected response headers.
- Associate path parameters, query string parameters, or header parameters in the method request of HTTP proxy or AWS service proxy.
- Select which data to send using integration with AWS services, such as Amazon DynamoDB or Lambda functions, or HTTP endpoints.

You can use mapping templates to transform your data. A *mapping template* is a script expressed in Velocity Template Language (VTL) and applied to the payload using JSONPath .

The following example shows input data, a mapping template, and output data for a transformation of the PetStore data.

Input data

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

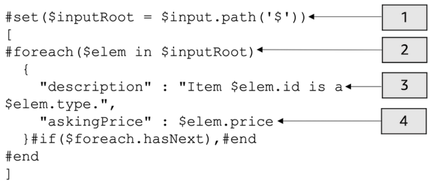
Mappin templa

```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a $elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end

#end
]
```

Output data

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

The following diagram shows details of this mapping template.



```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a
$elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end
#end
]
```

1. The `$inputRoot` variable represents the root object in the original JSON data from the previous section. Directives begin with the # symbol.
2. A `foreach` loop iterates though each object in the original JSON data.
3. The description is a concatenation of the Pet's `id` and `type` from the original JSON data.
4. `askingPrice` is the `price` is the price from the original JSON data.

PetStore mapping template

```
1 #set($inputRoot = $input.path('$'))
2 [
3 #foreach($elem in $inputRoot)
4  {
5    "description" : "Item $elem.id is a $elem.type.",
6    "askingPrice" : $elem.price
7  }#if($foreach.hasNext),#end
8 #end
9 ]
```

In this mapping template:

1. On line 1, the `$inputRoot` variable represents the root object in the original JSON data from the previous section. Directives begin with the # symbol.
2. On line 3, a `foreach` loop iterates through each object in the original JSON data.
3. On line 5, the `description` is a concatenation of the Pet's `id` and `type` from the original JSON data.

4. On line 6, `askingPrice` is the `price` is the price from the original JSON data.

For more information about the Velocity Template Language, see Apache Velocity - VTL Reference. For more information about JSONPath, see JSONPath - XPath for JSON.

The mapping template assumes that the underlying data is of a JSON object. It does not require that a model be defined for the data. However, a model for the output data allows preceding data to be returned as a language-specific object. For more information, see Understanding data models.

Complex mapping templates

You can also create more complicated mapping templates. The following example shows the concatenation of references and a cutoff of 100 to determine if a pet is affordable.

Input data

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

Mappin templa

```
#set($inputRoot = $input.path('$'))
#set($cheap = 100)
[
#foreach($elem in $inputRoot)
  {
#set($name = "${elem.type}number$elem.id")
    "name" : $name,
    "description" : "Item $elem.id is a $elem.type.",
    #if($elem.price > $cheap )#set ($afford = 'too much!') #{else}#set
 ($afford = $elem.price)#end
"askingPrice" : $afford
  }#if($foreach.hasNext),#end

#end
]
```

Output data

```
[
  {
    "name" : dognumber1,
    "description" : "Item 1 is a dog.",
    "askingPrice" : too much!
  },
  {
    "name" : catnumber2,
    "description" : "Item 2 is a cat.",
    "askingPrice" : too much!
  },
  {
    "name" : fishnumber3,
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

See the example photo album Photos example for a more complicated model.

Set up data transformations in API Gateway

This section shows how to set up mapping templates to transform integration requests and responses using the console and AWS CLI.

Topics

- Set up data transformation using the API Gateway console
- Set up data transformation using the AWS CLI
- Completed data transformation AWS CloudFormation template
- Next steps

Set up data transformation using the API Gateway console

In this tutorial, you will create an incomplete API and DynamoDB table using the following .zip file data-transformation-tutorial-console.zip. This incomplete API has a `/pets` resource with `GET` and `POST` methods.

- The `GET` method will get data from the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP endpoint. The output data will be transformed according to the mapping template in PetStore mapping template.
- The `POST` method will allow the user to `POST` pet information to a Amazon DynamoDB table using a mapping template.

Download and unzip the app creation template for AWS CloudFormation. You'll use this template to create a DynamoDB table to post pet information and an incomplete API. You will finish the rest of the steps in the API Gateway console.

To create an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create stack** and then choose **With new resources (standard)**.
3. For **Specify template**, choose **Upload a template file**.
4. Select the template that you downloaded.
5. Choose **Next**.
6. For **Stack name**, enter **`data-transformation-tutorial-console`** and then choose **Next**.
7. For **Configure stack options**, choose **Next**.
8. For **Capabilities**, acknowledge that AWS CloudFormation can create IAM resources in your account.
9. Choose **Submit**.

AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. When the status of your AWS CloudFormation stack is **CREATE_COMPLETE**, you're ready to move on to the next step.

To test the GET integration response

1. On the **Resources** tab of the AWS CloudFormation stack for **`data-transformation-tutorial-console`**, select the physical ID of your API.
2. In the main navigation pane, choose **Resources**, and then select the **GET** method.
3. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.

   The output of the test will show the following:

   ```
   [
     {
       "id": 1,
       "type": "dog",
       "price": 249.99
     },
     {
       "id": 2,
       "type": "cat",
       "price": 124.99
     },
     {
       "id": 3,
       "type": "fish",
       "price": 0.99
     }
   ]
   ```

   You will transform this output according to the mapping template in PetStore mapping template.

To transform the GET integration response

1. Choose the **Integration response** tab.

   Currently, there are no mapping templates defined, so the integration response will not be transformed.
2. For **Default - Response**, choose **Edit**.
3. Choose **Mapping templates**, and then do the following:

   a. Choose **Add mapping template**.

b. For **Content type**, enter **`application/json`**.

c. For **Template body**, enter the following:

```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a $elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end

#end
]
```

Choose **Save**.

To test the GET integration response

- Choose the **Test** tab, and then choose **Test**.

  The output of the test will show the transformed response.

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

To transform input data from the POST method

1. Choose the **POST** method.

2. Choose the **Integration request** tab, and then for **Integration request settings**, choose **Edit**.

   The AWS CloudFormation template has populated some of the integration request fields.

   - The integration type is AWS service.
   - The AWS service is DynamoDB.
   - The HTTP method is POST.
   - The Action is PutItem.
   - The Execution role allowing API Gateway to put an item into the DynamoDB table is data-transformation-tutorial-console-APIGatewayRole. AWS CloudFormation created this role to allow API Gateway to have the minimal permissions for interacting with DynamoDB.

   The name of the DynamoDB table has not been specified. You will specify the name in the following steps.

3. For **Request body passthrough**, select **Never**.

   This means that the API will reject data with Content-Types that do not have a mapping template.

4. Choose **Mapping templates**.

5. The **Content type** is set to application/json. This means a content types that are not application/json will be rejected by the API. For more information about the integration passthrough behaviors, see Integration passthrough behaviors

6. Enter the following code into the text editor.

```
{
    "TableName":"data-transformation-tutorial-console-ddb",
    "Item": {
        "id": {
            "N": $input.json("$.id")
        },
        "type": {
            "S": $input.json("$.type")
        },
        "price": {
            "N": $input.json("$.price")
        }
    }
}
```

This template specifies the table as `data-transformation-tutorial-console-ddb` and sets the items as `id`, `type`, and `price`. The items will come from the body of the `POST` method. You also can use a data model to help create a mapping template. For more information, see Use request validation in API Gateway.

7. Choose **Save** to save your mapping template.

To add a method and integration response from the POST method

The AWS CloudFormation created a blank method and integration response. You will edit this response to provide more information. For more information about how to edit responses, see Amazon API Gateway API request and response data mapping reference.

1. On the **Integration response** tab, for **Default - Response**, choose **Edit**.
2. Choose **Mapping templates**, and then choose **Add mapping template**.
3. For **Content-type**, enter **`application/json`**.
4. In the code editor, enter the following output mapping template to send an output message:

```
{ "message" : "Your response was recorded at $context.requestTime" }
```

For more information about context variables, see `$context` Variables for data models, authorizers, mapping templates, and CloudWatch access logging.

5. Choose **Save** to save your mapping template.

Test the POST method

Choose the **Test** tab. You might need to choose the right arrow button to show the tab.

1. In the request body, enter the following example.

```
{
          "id": "4",
          "type" : "dog",
          "price": "321"
}
```

2. Choose **Test**.

   The output should show your success message.

   You can open the DynamoDB console at https://console.aws.amazon.com/dynamodb/ to verify that the example item is in your table.

To delete an AWS CloudFormation stack

1. Open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Select your AWS CloudFormation stack.
3. Choose **Delete** and then confirm your choice.

Set up data transformation using the AWS CLI

In this tutorial, you will create an incomplete API and DynamoDB table using the following .zip file data-transformation-tutorial-cli.zip. This incomplete API has a `/pets` resource with a `GET` method integrated with the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP endpoint. You will create a `POST` method to connect to a DynamoDB table and use mapping templates to input data into a DynamoDB table.

- You will transform the output data according to the mapping template in PetStore mapping template.
- You will create a `POST` method to allow the user to `POST` pet information to a Amazon DynamoDB table using a mapping template.

To create an AWS CloudFormation stack

Download and unzip the app creation template for AWS CloudFormation.

To complete the following tutorial, you need the AWS Command Line Interface (AWS CLI) version 2.

For long commands, an escape character (\) is used to split a command over multiple lines.

Note

In Windows, some Bash CLI commands that you commonly use (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

1. Use the following command to create the AWS CloudFormation stack.

```
aws cloudformation create-stack --stack-name data-transformation-tutorial-cli
 --template-body file://data-transformation-tutorial-cli.zip --capabilities
 CAPABILITY_NAMED_IAM
```

2. AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. Use the following command to see the status of your AWS CloudFormation stack.

```
aws cloudformation describe-stacks --stack-name data-transformation-tutorial-cli
```

3. When the status of your AWS CloudFormation stack is `StackStatus: "CREATE_COMPLETE"`, use the following command to retrieve relevant output values for future steps.

```
 aws cloudformation describe-stacks --stack-name data-transformation-tutorial-cli
 --query "Stacks[*].Outputs[*].{OutputKey: OutputKey, OutputValue: OutputValue,
 Description: Description}"
```

The output values are the following:

- ApiRole, which is the role name that allows API Gateway to put items in the DynamoDB table. For this tutorial, the role name is `data-transformation-tutorial-cli-APIGatewayRole-ABCDEFG`.
- DDBTableName, which is the name of the DynamoDB table. For this tutorial, the table name is `data-transformation-tutorial-cli-ddb`
- ResourceId, which is the ID for the pets resource where the `GET` and `POST` methods are exposed. For this tutorial, the Resource ID is `efg456`
- ApiId, which is the ID for the API. For this tutorial, the API ID is `abc123`.

To test the GET method before data transformation

- Use the following command to test the GET method.

```
aws apigateway test-invoke-method --rest-api-id abc123 \
          --resource-id efg456 \
          --http-method GET
```

The output of the test will show the following.

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

You will transform this output according to the mapping template in PetStore mapping template.

To transform the GET integration response

- Use the following command to update the integration response for the GET method. Replace the *`rest-api-id`* and *`resource-id`* with your values.

Use the following command to create the integration response.

```
aws apigateway put-integration-response --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method GET \
  --status-code 200 \
  --selection-pattern "" \
  --response-templates '{"application/json": "#set($inputRoot = $input.path(\"$
\"))\n[\n#foreach($elem in $inputRoot)\n {\n  \"description\": \"Item $elem.id is a
 $elem.type\",\n  \"askingPrice\": \"$elem.price\"\n }#if($foreach.hasNext),#end\n
\n#end\n]"}'
```

To test the GET method

- Use the following command to test the GET method.

```
aws apigateway test-invoke-method --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method GET \
```

The output of the test will show the transformed response.

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

To create a POST method

1. Use the following command to create a new method on the `/pets` resource.

```
aws apigateway put-method --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --authorization-type "NONE" \
```

This method will allow you to send pet information to the DynamoDB table that your created in the AWS CloudFormation stack.

2. Use the following command to create an AWS service integration on the POST method.

```
aws apigateway put-integration --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --type AWS \
  --integration-http-method POST \
  --uri "arn:aws:apigateway:us-east-2:dynamodb:action/PutItem" \
  --credentials arn:aws:iam::111122223333:role/data-transformation-tutorial-cli-
APIGatewayRole-ABCDEFG \
  --request-templates '{"application/json":"{\"TableName\":\"data-transformation-
tutorial-cli-ddb\",\"Item\":{\"id\":{\"N\":$input.json(\"$.id\")},\"type\":{\"S\":
$input.json(\"$.type\")},\"price\":{\"N\":$input.json(\"$.price\")} }}"}'
```

3. Use the following command to create a method response for a successful call of the POST method.

```
aws apigateway put-method-response --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --status-code 200
```

4. Use the following command to create an integration response for the successful call of the POST method.

```
aws apigateway put-integration-response --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --status-code 200 \
  --selection-pattern "" \
  --response-templates '{"application/json": "{\"message\": \"Your response was
 recorded at $context.requestTime\"}"}'
```

To test the POST method

- Use the following command to test the POST method.

```
aws apigateway test-invoke-method --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --body '{\"id\": \"4\", \"type\": \"dog\", \"price\": \"321\"}'
```

The output will show the successful message.

To delete an AWS CloudFormation stack

- Use the following command to delete your AWS CloudFormation resources.

```
aws cloudformation delete-stack  --stack-name data-transformation-tutorial-cli
```

Completed data transformation AWS CloudFormation template

The following example is a completed AWS CloudFormation template, which creates an API and a DynamoDB table with a `/pets` resource with `GET` and `POST` methods.

- The `GET` method will get data from the `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP endpoint. The output data will be transformed according to the mapping template in PetStore mapping template.
- The `POST` method will allow the user to `POST` pet information to a DynamoDB table using a mapping template.

```
AWSTemplateFormatVersion: 2010-09-09
Description: A completed Amazon API Gateway REST API that uses non-proxy integration
 to POST to an Amazon DynamoDB table and non-proxy integration to GET transformed pets
 data.
Parameters:
  StageName:
    Type: String
    Default: v1
    Description: Name of API stage.
Resources:
  DynamoDBTable:
    Type: 'AWS::DynamoDB::Table'
    Properties:
      TableName: !Sub data-transformation-tutorial-complete
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: N
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
  APIGatewayRole:
    Type: 'AWS::IAM::Role'
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Action:
              - 'sts:AssumeRole'
            Effect: Allow
            Principal:
              Service:
                - apigateway.amazonaws.com
      Policies:
        - PolicyName: APIGatewayDynamoDBPolicy
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:PutItem'
                Resource: !GetAtt DynamoDBTable.Arn
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: data-transformation-complete-api
      ApiKeySourceType: HEADER
  PetsResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'pets'
  PetsMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: HTTP
        Credentials: !GetAtt APIGatewayRole.Arn
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
        PassthroughBehavior: WHEN_NO_TEMPLATES
        IntegrationResponses:
          - StatusCode: '200'
            ResponseTemplates:
              application/json: "#set($inputRoot = $input.path(\"$
\"))\n[\n#foreach($elem in $inputRoot)\n {\n  \"description\": \"Item $elem.id is a
 $elem.type\",\n  \"askingPrice\": \"$elem.price\"\n }#if($foreach.hasNext),#end\n
\n#end\n]"
      MethodResponses:
        - StatusCode: '200'
  PetsMethodPost:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: POST
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: AWS
        Credentials: !GetAtt APIGatewayRole.Arn
        IntegrationHttpMethod: POST
        Uri: arn:aws:apigateway:us-west-1:dynamodb:action/PutItem
        PassthroughBehavior: NEVER
        RequestTemplates:
          application/json: "{\"TableName\":\"data-transformation-tutorial-complete
\",\"Item\":{\"id\":{\"N\":$input.json(\"$.id\")},\"type\":{\"S\":$input.json(\"$.type
\")},\"price\":{\"N\":$input.json(\"$.price\")} }}"
        IntegrationResponses:
          - StatusCode: 200
            ResponseTemplates:
              application/json: "{\"message\": \"Your response was recorded at
 $context.requestTime\"}"
```

```
      MethodResponses:
        - StatusCode: '200'

  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - PetsMethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: !Sub '${StageName}'
Outputs:
  ApiId:
    Description: API ID for CLI commands
    Value: !Ref Api
  ResourceId:
    Description: /pets resource ID for CLI commands
    Value: !Ref PetsResource
  ApiRole:
    Description: Role ID to allow API Gateway to put and scan items in DynamoDB table
    Value: !Ref APIGatewayRole
  DDBTableName:
    Description: DynamoDB table name
    Value: !Ref DynamoDBTable
```

Next steps

To explore more complex mapping templates, see the following examples:

- See more complex models and mapping templates with the example photo album Photos example.
- For more information about models, see Understanding data models.
- For information about how to map different response code outputs, Set up request and response data mappings using the API Gateway console.
- For information about how to set data mappings from an API's method request data, API Gateway mapping template and access logging variable reference.

Use a mapping template to override an API's request and response parameters and status codes

Standard API Gateway parameter and response code mapping templates allow you to map parameters one-to-one and map a family of integration response status codes (matched by a regular expression) to a single response status code. Mapping template overrides provides you with the flexibility to perform many-to-one parameter mappings; override parameters after standard API Gateway mappings have been applied; conditionally map parameters based on body content or other parameter values; programmatically create new parameters on the fly; and override status codes returned by your integration endpoint. Any type of request parameter, response header, or response status code may be overridden.

Following are example uses for a mapping template override:

- To create a new header (or overwrite an existing header) as a concatenation of two parameters
- To override the response code to a success or failure code based on the contents of the body
- To conditionally remap a parameter based on its contents or the contents of some other parameter
- To iterate over the contents of a json body and remap key value pairs to headers or query strings

To create a mapping template override, use one or more of the following `$context` variables in a mapping template:

Request body mapping template	Response body mapping template
`$context.requestOverride.header.` *`header_name`*	`$context.responseOverride.header.` *`header_name`*
`$context.requestOverride.path.` *`path_name`*	`$context.responseOverride.status`
`$context.requestOverride.querystring.` *`querystring_name`*	

Note

Mapping template overrides cannot be used with proxy integration endpoints, which lack data mappings. For more information about integration types, see Choose an API Gateway API integration type.

Important

Overrides are final. An override may only be applied to each parameter once. Trying to override the same parameter multiple times will result in 5XX responses from Amazon API Gateway. If you must override the same parameter multiple times throughout the template, we recommend creating a variable and applying the override at the end of the template. Note that the template is applied only after the entire template is parsed. See Tutorial: Override an API's request parameters and headers with the API Gateway console.

The following tutorials show how to create and test a mapping template override in the API Gateway console. These tutorials use the PetStore sample API as a starting point. Both tutorials assume that you have already created the PetStore sample API.

Topics

Tutorial: Override an API's response status code with the API Gateway console

To retrieve a pet using the PetStore sample API, you use the API method request of `GET /pets/{petId}`, where `{petId}` is a path parameter that can take a number at run time.

In this tutorial, you'll override this `GET` method's response code by creating a mapping template that maps `$context.responseOverride.status` to `400` when an error condition is detected.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Under **APIs**, choose the PetStore API, and then choose **Resources**.

3. In the **Resources** tree, choose the GET method under /{petId}.
4. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
5. For **petId**, enter **-1**, and then choose **Test**.

   In the results, you'll notice two things:

   First, the **Response body** indicates an out-of-range error:

   ```
   {
     "errors": [
       {
         "key": "GetPetRequest.petId",
         "message": "The value is out of range."
       }
     ]
   }
   ```

   Second, the last line under **Log** box ends with: `Method completed with status: 200`.
6. On the **Integration response** tab, for the **Default - Response**, choose **Edit**.
7. Choose **Mapping templates**.
8. Choose **Add mapping template**.
9. For **Content type**, enter **`application/json`**.
10. For **Template body**, enter the following:

    ```
    #set($inputRoot = $input.path('$'))
    $input.json("$")
    #if($inputRoot.toString().contains("error"))
    #set($context.responseOverride.status = 400)
    #end
    ```
11. Choose **Save**.
12. Choose the **Test** tab.
13. For **petId**, enter **-1**.
14. In the results, the **Response Body** indicates an out-of-range error:

    ```
    {
      "errors": [
    ```

```
    {
      "key": "GetPetRequest.petId",
      "message": "The value is out of range."
    }
  ]
}
```

However, the last line under **Logs** box now ends with: `Method completed with status: 400`.

Tutorial: Override an API's request parameters and headers with the API Gateway console

In this tutorial, you'll override the GET method's request header code by creating a mapping template that maps `$context.requestOverride.header.`*`header_name`* to a new header that combines two other headers.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Under **APIs**, choose the PetStore API.
3. In the **Resources** tree, choose the GET method under `/pet`.
4. On the **Method request** tab, for **Method request settings**, choose **Edit**.
5. Choose **HTTP request headers**, and then choose **Add header**.
6. For **Name**, enter **`header1`**.
7. Choose **Add header**, and then create a second header called **`header2`**.
8. Choose **Save**.
9. On the **Integration request** tab, for **Integration request settings**, choose **Edit**.
10. For **Request body passthrough**, select **When there are no templates defined (recommended)**.
11. Choose **Mapping templates**, and then do the following:

    a. Choose **Add mapping template**.

    b. For **Content type**, enter **`application/json`**.

    c. For **Template body**, enter the following:

    ```
    #set($header1Override = "foo")
    #set($header3Value = "$input.params('header1')$input.params('header2')")
    $input.json("$")
    ```

```
#set($context.requestOverride.header.header3 = $header3Value)
#set($context.requestOverride.header.header1 = $header1Override)
#set($context.requestOverride.header.multivalueheader=[$header1Override,
 $header3Value])
```

12. Choose **Save**.
13. Choose the **Test** tab.
14. Under **Headers** for **{pets}**, copy the following code:

    ```
    header1:header1Val
    header2:header2Val
    ```

15. Choose **Test**.

    In the Log, you should see an entry that includes this text:

    ```
    Endpoint request headers: {header3=header1Valheader2Val,
    header2=header2Val, header1=foo, x-amzn-apigateway-api-id=<api-id>,
    Accept=application/json, multivalueheader=foo,header1Valheader2Val}
    ```

Examples: Override an API's request parameters and headers with the API Gateway CLI

The following CLI example shows how to use the `put-integration` command to override a response code:

```
aws apigateway put-integration --rest-api-id <API_ID> --resource-
id <PATH_TO_RESOURCE_ID> --http-method <METHOD>
            --type <INTEGRATION_TYPE> --request-templates <REQUEST_TEMPLATE_MAP>
```

where *<REQUEST_TEMPLATE_MAP>* is a map from content type to a string of the template to apply. The structure of that map is as follows:

```
Content_type1=template_string,Content_type2=template_string
```

or, in JSON syntax:

```
{"content_type1": "template_string"
...}
```

The following example shows how to use the `put-integration-response` command to override an API's response code:

```
aws apigateway put-integration-response --rest-api-id <API_ID> --resource-
id <PATH_TO_RESOURCE_ID> --http-method <METHOD>
            --status-code <STATUS_CODE> --response-templates <RESPONSE_TEMPLATE_MAP>
```

where *<RESPONSE_TEMPLATE_MAP>* has the same format as *<REQUEST_TEMPLATE_MAP>* above.

Example: Override an API's request parameters and headers using the SDK for JavaScript

The following example shows how to use the `put-integration` command to override a response code:

Request:

```
var params = {
  httpMethod: 'STRING_VALUE', /* required */
  resourceId: 'STRING_VALUE', /* required */
  restApiId: 'STRING_VALUE', /* required */
  type: HTTP | AWS | MOCK | HTTP_PROXY | AWS_PROXY, /* required */
  requestTemplates: {
    '<Content_type>': 'TEMPLATE_STRING',
    /* '<String>': ... */
  },
};
apigateway.putIntegration(params, function(err, data) {
  if (err) console.log(err, err.stack); // an error occurred
  else     console.log(data);           // successful response
});
```

Response:

```
var params = {
  httpMethod: 'STRING_VALUE', /* required */
  resourceId: 'STRING_VALUE', /* required */
  restApiId: 'STRING_VALUE', /* required */
  statusCode: 'STRING_VALUE', /* required */
  responseTemplates: {
    '<Content_type>': 'TEMPLATE_STRING',
    /* '<String>': ... */
  },
};
apigateway.putIntegrationResponse(params, function(err, data) {
  if (err) console.log(err, err.stack); // an error occurred
  else     console.log(data);           // successful response
});
```

Set up request and response data mappings using the API Gateway console

To use the API Gateway console to define the API's integration request/response, follow these instructions.

> **Note**
>
> These instructions assume you have already completed the steps in Set up an API integration request using the API Gateway console.

1. In the **Resources** pane, choose your method.
2. On the **Integration request** tab, under **Integration request settings**, choose **Edit**.
3. Choose an option for **Request body passthrough** to configure how the method request body of an unmapped content type will be passed through the integration request without transformation to the Lambda function, HTTP proxy, or AWS service proxy. There are three options:

    - Choose **When no template matches the request content-type header** if you want the method request body to pass through the integration request to the backend without transformation when the method request content type does not match any content types associated with the mapping templates, as defined in the next step.

      > **Note**
      >
      > When calling the API Gateway API, you choose this option by setting `WHEN_NO_MATCH` as the `passthroughBehavior` property value on the Integration resource.

    - Choose **When there are no templates defined (recommended)** if you want the method request body to pass through the integration request to the backend without transformation when no mapping template is defined in the integration request. If a template is defined when this option is selected, the method request of an unmapped content type will be rejected with an HTTP 415 Unsupported Media Type response.

> **Note**
>
> When calling the API Gateway API, you choose this option by setting `WHEN_NO_TEMPLATE` as the `passthroughBehavior` property value on the Integration resource.

- Choose **Never** if you do not want the method request to pass through when either the method request content type does not match any content type associated with the mapping templates defined in the integration request or no mapping template is defined in the integration request. The method request of an unmapped content type will be rejected with an HTTP 415 Unsupported Media Type response.

> **Note**
>
> When calling the API Gateway API, you choose this option by setting `NEVER` as the `passthroughBehavior` property value on the Integration resource.

For more information about the integration passthrough behaviors, see Integration passthrough behaviors.

4. For an HTTP proxy or an AWS service proxy, to associate a path parameter, a query string parameter, or a header parameter defined in the integration request with a corresponding path parameter, query string parameter, or header parameter in the method request of the HTTP proxy or AWS service proxy, do the following:

    a. Choose **URL path parameters**, **URL query string parameters**, or **HTTP request headers** respectively, and then choose **Add path**, **Add query string**, or **Add header**, respectively.

    b. For **Name**, type the name of the path parameter, query string parameter, or header parameter in the HTTP proxy or AWS service proxy.

    c. For **Mapped from**, enter the mapping value for the path parameter, query string parameter, or header parameter. Use one of the following formats:

    - **`method.request.path.`*`parameter-name`*** for a path parameter named *`parameter-name`* as defined in the **Method request** page.

- **method.request.querystring.*parameter-name*** for a query string parameter named *parameter-name* as defined in the **Method request** page.
- **method.request.multivaluequerystring.*parameter-name*** for a multi-valued query string parameter named *parameter-name* as defined in the **Method request** page.
- **method.request.header.*parameter-name*** for a header parameter named *parameter-name* as defined in the **Method request** page.

  Alternatively, you can set a literal string value (enclosed by a pair of single quotes) to an integration header.
- **method.request.multivalueheader.*parameter-name*** for a multi-valued header parameter named *parameter-name* as defined in the **Method request** page.

d. To add another parameter, choose the **Add** button.

5. To add a mapping template, choose **Mapping templates**.
6. To define a mapping template for an incoming request, choose **Add mapping template**. For **Content type**, enter a content type (e.g., **application/json**). Then, enter the mapping template manually or choose **Generate template** to create one from a model template. For more information, see Understanding mapping templates.
7. Choose **Save**.
8. You can map an integration response from the backend to a method response of the API returned to the calling app. This includes returning to the client selected response headers from the available ones from the backend, transforming the data format of the backend response payload to an API-specified format. You can specify such mapping by configuring **Method response** and **Integration responses**.

   To have the method receive a custom response data format based on the HTTP status code returned by the Lambda function, HTTP proxy, or AWS service proxy, do the following:

   a. Choose **Integration responses**. Choose either **Edit** on the **Default - Response** , to specify settings for a 200 HTTP response code from the method, or choose **Create response** to specify settings for any other HTTP response status code from the method.

   b. For **Lambda error regex** (for a Lambda function) or **HTTP status regex** (for an HTTP proxy or AWS service proxy), enter a regular expression to specify which Lambda function error strings (for a Lambda function) or HTTP response status codes (for an HTTP proxy or AWS service proxy) map to this output mapping. For example, to map all 2xx HTTP response status codes from an HTTP proxy to this output mapping, type "**`2\d{2}`**" for **HTTP status regex**. To return an error message containing "Invalid Request" from a Lambda function to a `400 Bad Request` response, enter "**`.*Invalid request.*`**" as the **Lambda error regex** expression. On the other hand, to return `400 Bad Request` for all unmapped error messages from Lambda, enter "**`(\n|.)+`**" in **Lambda error regex**. This last regular expression can be used for the default error response of an API.

> **Note**
>
> API Gateway uses Java pattern-style regexes for response mapping. For more information, see Pattern in the Oracle documentation.
> The error patterns are matched against the entire string of the `errorMessage` property in the Lambda response, which is populated by `callback(errorMessage)` in Node.js or by `throw new MyException(errorMessage)` in Java. Also, escaped characters are unescaped before the regular expression is applied.
> If you use '.+' as the selection pattern to filter responses, be aware that it may not match a response containing a newline ('\n') character.

c. If enabled, for **Method response status**, select the HTTP response status code you defined on the **Method response** page.

d. For **Header mappings**, for each header you defined for the HTTP response status code on the **Method response** page, specify a mapping value. For **Mapping value**, use one of the following formats:

- **`integration.response.multivalueheaders.`*`header-name`*** where *`header-name`* is the name of a multi-valued response header from the backend.

  For example, to return the backend response's `Date` header as an API method's response's `Timestamp` header, the **Response header** column will contain a **Timestamp** entry, and the associated **Mapping value** should be set to **integration.response.multivalueheaders.Date**.

- **`integration.response.header.`*`header-name`*** where *`header-name`* is the name of a single-valued response header from the backend.

  For example, to return the backend response's `Date` header as an API method's response's `Timestamp` header, the **Response header** column will contain a **Timestamp** entry, and the associated **Mapping value** should be set to **integration.response.header.Date**.

e. Choose **Mapping templates**, and then choose **Add mapping template**. In the **Content type** box, enter the content type of the data that will be passed from the Lambda function, HTTP proxy, or AWS service proxy to the method. Then, enter the mapping template manually or choose **Generate template** to create one from a model template. For more information, see Understanding mapping templates.

f. Choose **Save**.

Models and mapping template examples

The following sections provide examples of models and mapping templates that could be used as a starting point for your own APIs in API Gateway. For more information about models and mapping templates in API Gateway, see PetStore mapping template.

Topics

- Photos example (API Gateway models and mapping templates)
- News article example (API Gateway models and mapping templates)
- Sales invoice example (API Gateway models and mapping templates)
- Employee record example (API Gateway models and mapping templates)

Photos example (API Gateway models and mapping templates)

The following example shows a photo album API in API Gateway. We provide an example data transformation, additional models, and mapping templates. For more information about data transformations, see Understanding mapping templates For more information about models and mapping templates in API Gateway, see PetStore mapping template.

Topics

- Example data transformation
- Input model for photo data
- Output model for photo data
- Input mapping template for photo data

Example data transformation

The following example shows how you can transform input data about photos by using a Velocity Template Language (VTL) mapping template. For more information about the Velocity Template Language, see Apache Velocity - VTL Reference.

Input data

```
{
  "photos": {
    "page": 1,
    "pages": "1234",
    "perpage": 100,
    "total": "123398",
    "photo": [
      {
        "id": "12345678901",
        "owner": "23456789@A12",
        "photographer_first_name" : "Saanvi",
        "photographer_last_name" : "Sarkar",
        "secret": "abc123d456",
        "server": "1234",
        "farm": 1,
        "title": "Sample photo 1",
        "ispublic": true,
        "isfriend": false,
        "isfamily": false
      },
      {
        "id": "23456789012",
        "owner": "34567890@B23",
        "photographer_first_name" : "Richard",
        "photographer_last_name" : "Roe",
        "secret": "bcd234e567",
        "server": "2345",
        "farm": 2,
        "title": "Sample photo 2",
        "ispublic": true,
        "isfriend": false,
        "isfamily": false
      }
    ]
  }
}
```

Output mappin templa

```
#set($inputRoot = $input.path('$'))
{
  "photos": [
#foreach($elem in $inputRoot.photos.photo)
    {
      "id": "$elem.id",
      "photographedBy": "$elem.photographer_first_name $elem.pho
tographer_last_name",
      "title": "$elem.title",
      "ispublic": $elem.ispublic,
      "isfriend": $elem.isfriend,
      "isfamily": $elem.isfamily
    }#if($foreach.hasNext),#end

#end
  ]
}
```

Output data

```
{
  "photos": [
    {
      "id": "12345678901",
      "photographedBy": "Saanvi Sarkar",
      "title": "Sample photo 1",
      "ispublic": true,
      "isfriend": false,
      "isfamily": false
    },
    {
      "id": "23456789012",
      "photographedBy": "Richard Roe",
      "title": "Sample photo 2",
      "ispublic": true,
      "isfriend": false,
      "isfamily": false
    }
  ]
}
```

Input model for photo data

You can define a model for your input data. This input model requires that you upload one photo, and it specifies a minimum of 10 photos per page. You can use this input model to generate an SDK or to turn on a request validation for your API.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosInputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "object",
      "required" : [
      "photo"
      ],
      "properties": {
        "page": { "type": "integer" },
        "pages": { "type": "string" },
        "perpage": { "type": "integer", "minimum" : 10 },
        "total": { "type": "string" },
        "photo": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "id": { "type": "string" },
              "owner": { "type": "string" },
              "photographer_first_name" : {"type" : "string"},
              "photographer_last_name" : {"type" : "string"},
              "secret": { "type": "string" },
              "server": { "type": "string" },
              "farm": { "type": "integer" },
              "title": { "type": "string" },
              "ispublic": { "type": "boolean" },
              "isfriend": { "type": "boolean" },
              "isfamily": { "type": "boolean" }
            }
          }
        }
      }
    }
  }
}
```

Output model for photo data

You can define a model for your output data. You can use this model for a method response model, which is necessary when you generate a strongly typed SDK for the API. This causes the output to be cast into an appropriate class in Java or Objective-C.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosOutputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string" },
          "photographedBy": { "type": "string" },
          "title": { "type": "string" },
          "ispublic": { "type": "boolean" },
          "isfriend": { "type": "boolean" },
          "isfamily": { "type": "boolean" }
        }
      }
    }
  }
}
```

Input mapping template for photo data

You can define a mapping template to modify input data. You can modify input data for further function integration or integration responses.

```
#set($inputRoot = $input.path('$'))
{
  "photos": {
    "page": $inputRoot.photos.page,
    "pages": "$inputRoot.photos.pages",
    "perpage": $inputRoot.photos.perpage,
    "total": "$inputRoot.photos.total",
    "photo": [
#foreach($elem in $inputRoot.photos.photo)
      {
        "id": "$elem.id",
        "owner": "$elem.owner",
        "photographer_first_name" : "$elem.photographer_first_name",
        "photographer_last_name" : "$elem.photographer_last_name",
        "secret": "$elem.secret",
        "server": "$elem.server",
        "farm": $elem.farm,
        "title": "$elem.title",
        "ispublic": $elem.ispublic,
        "isfriend": $elem.isfriend,
        "isfamily": $elem.isfamily
      }#if($foreach.hasNext),#end

#end
    ]
  }
}
```

News article example (API Gateway models and mapping templates)

The following sections provide examples of models and mapping templates that could be used for a sample news article API in API Gateway. For more information about models and mapping templates in API Gateway, see PetStore mapping template.

Topics

Original data (news article example)

The following is the original JSON data for the news article example:

```
{
  "count": 1,
  "items": [
    {
      "last_updated_date": "2015-04-24",
      "expire_date": "2016-04-25",
      "author_first_name": "John",
      "description": "Sample Description",
      "creation_date": "2015-04-20",
      "title": "Sample Title",
      "allow_comment": "1",
      "author": {
        "last_name": "Doe",
        "email": "johndoe@example.com",
        "first_name": "John"
      },
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "version": "1",
      "author_last_name": "Doe",
      "parent_id": 2345678901,
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

Input model (news article example)

The following is the input model that corresponds to the original JSON data for the news article example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "NewsArticleInputModel",
  "type": "object",
  "properties": {
    "count": { "type": "integer" },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "last_updated_date": { "type": "string" },
     "expire_date": { "type": "string" },
     "author_first_name": { "type": "string" },
     "description": { "type": "string" },
     "creation_date": { "type": "string" },
     "title": { "type": "string" },
     "allow_comment": { "type": "string" },
     "author": {
      "type": "object",
      "properties": {
       "last_name": { "type": "string" },
       "email": { "type": "string" },
       "first_name": { "type": "string" }
      }
     },
     "body": { "type": "string" },
     "publish_date": { "type": "string" },
     "version": { "type": "string" },
     "author_last_name": { "type": "string" },
     "parent_id": { "type": "integer" },
     "article_url": { "type": "string" }
    }
   }
  },
  "version": { "type": "integer" }
 }
}
```

Input mapping template (news article example)

The following is the input mapping template that corresponds to the original JSON data for the news article example:

```
#set($inputRoot = $input.path('$'))
{
 "count": $inputRoot.count,
 "items": [
#foreach($elem in $inputRoot.items)
  {
   "last_updated_date": "$elem.last_updated_date",
   "expire_date": "$elem.expire_date",
   "author_first_name": "$elem.author_first_name",
   "description": "$elem.description",
   "creation_date": "$elem.creation_date",
   "title": "$elem.title",
   "allow_comment": "$elem.allow_comment",
   "author": {
     "last_name": "$elem.author.last_name",
     "email": "$elem.author.email",
     "first_name": "$elem.author.first_name"
   },
   "body": "$elem.body",
   "publish_date": "$elem.publish_date",
   "version": "$elem.version",
   "author_last_name": "$elem.author_last_name",
   "parent_id": $elem.parent_id,
   "article_url": "$elem.article_url"
  }#if($foreach.hasNext),#end

#end
 ],
 "version": $inputRoot.version
}
```

Transformed data (news article example)

The following is one example of how the original news article example JSON data could be transformed for output:

```
{
  "count": 1,
  "items": [
    {
      "creation_date": "2015-04-20",
      "title": "Sample Title",
      "author": "John Doe",
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

Output model (news article example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "NewsArticleOutputModel",
  "type": "object",
  "properties": {
    "count": { "type": "integer" },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "creation_date": { "type": "string" },
          "title": { "type": "string" },
          "author": { "type": "string" },
          "body": { "type": "string" },
          "publish_date": { "type": "string" },
          "article_url": { "type": "string" }
        }
      }
    },
    "version": { "type": "integer" }
  }
}
```

Output mapping template (news article example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "count": $inputRoot.count,
  "items": [
#foreach($elem in $inputRoot.items)
    {
      "creation_date": "$elem.creation_date",
      "title": "$elem.title",
      "author": "$elem.author.first_name $elem.author.last_name",
      "body": "$elem.body",
      "publish_date": "$elem.publish_date",
      "article_url": "$elem.article_url"
    }#if($foreach.hasNext),#end
#end
  ],
  "version": $inputRoot.version
}
```

Sales invoice example (API Gateway models and mapping templates)

The following sections provide examples of models and mapping templates that could be used for a sample sales invoice API in API Gateway. For more information about models and mapping templates in API Gateway, see PetStore mapping template.

Topics

Original data (sales invoice example)

The following is the original JSON data for the sales invoice example:

```
{
  "DueDate": "2013-02-15",
  "Balance": 1990.19,
  "DocNumber": "SAMP001",
  "Status": "Payable",
  "Line": [
    {
      "Description": "Sample Expense",
      "Amount": 500,
      "DetailType": "ExpenseDetail",
      "ExpenseDetail": {
        "Customer": {
          "value": "ABC123",
          "name": "Sample Customer"
        },
        "Ref": {
          "value": "DEF234",
          "name": "Sample Construction"
        },
        "Account": {
          "value": "EFG345",
          "name": "Fuel"
        },
        "LineStatus": "Billable"
      }
    }
  ],
  "Vendor": {
    "value": "GHI456",
    "name": "Sample Bank"
  },
  "APRef": {
    "value": "HIJ567",
    "name": "Accounts Payable"
  },
  "TotalAmt": 1990.19
}
```

Input model (sales invoice example)

The following is the input model that corresponds to the original JSON data for the sales invoice example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "InvoiceInputModel",
  "type": "object",
  "properties": {
    "DueDate": { "type": "string" },
    "Balance": { "type": "number" },
    "DocNumber": { "type": "string" },
    "Status": { "type": "string" },
    "Line": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "Description": { "type": "string" },
          "Amount": { "type": "integer" },
          "DetailType": { "type": "string" },
          "ExpenseDetail": {
            "type": "object",
            "properties": {
              "Customer": {
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "Ref": {
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "Account": {
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "LineStatus": { "type": "string" }
            }
          }
        }
      }
    },
    "Vendor": {
      "type": "object",
      "properties": {
        "value": { "type": "string" },
        "name": { "type": "string" }
      }
    },
    "APRef": {
      "type": "object",
      "properties": {
        "value": { "type": "string" },
        "name": { "type": "string" }
      }
    },
    "TotalAmt": { "type": "number" }
  }
}
```

Input mapping template (sales invoice example)

The following is the input mapping template that corresponds to the original JSON data for the sales invoice example:

```
#set($inputRoot = $input.path('$'))
{
  "DueDate": "$inputRoot.DueDate",
  "Balance": $inputRoot.Balance,
  "DocNumber": "$inputRoot.DocNumber",
  "Status": "$inputRoot.Status",
  "Line": [
#foreach($elem in $inputRoot.Line)
    {
      "Description": "$elem.Description",
      "Amount": $elem.Amount,
      "DetailType": "$elem.DetailType",
      "ExpenseDetail": {
        "Customer": {
          "value": "$elem.ExpenseDetail.Customer.value",
          "name": "$elem.ExpenseDetail.Customer.name"
        },
        "Ref": {
          "value": "$elem.ExpenseDetail.Ref.value",
          "name": "$elem.ExpenseDetail.Ref.name"
        },
        "Account": {
          "value": "$elem.ExpenseDetail.Account.value",
          "name": "$elem.ExpenseDetail.Account.name"
        },
        "LineStatus": "$elem.ExpenseDetail.LineStatus"
      }
    }#if($foreach.hasNext),#end

#end
  ],
  "Vendor": {
    "value": "$inputRoot.Vendor.value",
    "name": "$inputRoot.Vendor.name"
  },
  "APRef": {
    "value": "$inputRoot.APRef.value",
    "name": "$inputRoot.APRef.name"
  },
  "TotalAmt": $inputRoot.TotalAmt
}
```

Transformed data (sales invoice example)

The following is one example of how the original sales invoice example JSON data could be transformed for output:

```
{
  "DueDate": "2013-02-15",
  "Balance": 1990.19,
  "DocNumber": "SAMP001",
  "Status": "Payable",
  "Line": [
    {
      "Description": "Sample Expense",
      "Amount": 500,
      "DetailType": "ExpenseDetail",
      "Customer": "ABC123 (Sample Customer)",
      "Ref": "DEF234 (Sample Construction)",
      "Account": "EFG345 (Fuel)",
      "LineStatus": "Billable"
    }
  ],
  "TotalAmt": 1990.19
}
```

Output model (sales invoice example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "InvoiceOutputModel",
  "type": "object",
  "properties": {
    "DueDate": { "type": "string" },
    "Balance": { "type": "number" },
    "DocNumber": { "type": "string" },
    "Status": { "type": "string" },
    "Line": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "Description": { "type": "string" },
          "Amount": { "type": "integer" },
          "DetailType": { "type": "string" },
          "Customer": { "type": "string" },
          "Ref": { "type": "string" },
          "Account": { "type": "string" },
          "LineStatus": { "type": "string" }
        }
      }
    },
    "TotalAmt": { "type": "number" }
  }
}
```

Output mapping template (sales invoice example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "DueDate": "$inputRoot.DueDate",
  "Balance": $inputRoot.Balance,
  "DocNumber": "$inputRoot.DocNumber",
  "Status": "$inputRoot.Status",
  "Line": [
#foreach($elem in $inputRoot.Line)
    {
      "Description": "$elem.Description",
      "Amount": $elem.Amount,
      "DetailType": "$elem.DetailType",
      "Customer": "$elem.ExpenseDetail.Customer.value
 ($elem.ExpenseDetail.Customer.name)",
      "Ref": "$elem.ExpenseDetail.Ref.value ($elem.ExpenseDetail.Ref.name)",
      "Account": "$elem.ExpenseDetail.Account.value
 ($elem.ExpenseDetail.Account.name)",
      "LineStatus": "$elem.ExpenseDetail.LineStatus"
    }#if($foreach.hasNext),#end

#end
  ],
  "TotalAmt": $inputRoot.TotalAmt
}
```

Employee record example (API Gateway models and mapping templates)

The following sections provide examples of models and mapping templates that can be used for a sample employee record API in API Gateway. For more information about models and mapping templates in API Gateway, see PetStore mapping template.

Topics

- Original data (employee record example)
- Input model (employee record example)
- Input mapping template (employee record example)
- Transformed data (employee record example)
- Output model (employee record example)
- Output mapping template (employee record example)

Original data (employee record example)

The following is the original JSON data for the employee record example:

```
{
  "QueryResponse": {
    "maxResults": "1",
    "startPosition": "1",
    "Employee": {
      "Organization": "false",
      "Title": "Mrs.",
      "GivenName": "Jane",
      "MiddleName": "Lane",
      "FamilyName": "Doe",
      "DisplayName": "Jane Lane Doe",
      "PrintOnCheckName": "Jane Lane Doe",
      "Active": "true",
      "PrimaryPhone": { "FreeFormNumber": "505.555.9999" },
      "PrimaryEmailAddr": { "Address": "janedoe@example.com" },
      "EmployeeType": "Regular",
      "status": "Synchronized",
      "Id": "ABC123",
      "SyncToken": "1",
      "MetaData": {
        "CreateTime": "2015-04-26T19:45:03Z",
        "LastUpdatedTime": "2015-04-27T21:48:23Z"
      },
      "PrimaryAddr": {
        "Line1": "123 Any Street",
        "City": "Any City",
        "CountrySubDivisionCode": "WA",
        "PostalCode": "01234"
      }
    }
  },
  "time": "2015-04-27T22:12:32.012Z"
}
```

Input model (employee record example)

The following is the input model that corresponds to the original JSON data for the employee record example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "EmployeeInputModel",
  "type": "object",
  "properties": {
    "QueryResponse": {
      "type": "object",
      "properties": {
        "maxResults": { "type": "string" },
        "startPosition": { "type": "string" },
        "Employee": {
          "type": "object",
          "properties": {
            "Organization": { "type": "string" },
            "Title": { "type": "string" },
            "GivenName": { "type": "string" },
            "MiddleName": { "type": "string" },
            "FamilyName": { "type": "string" },
            "DisplayName": { "type": "string" },
            "PrintOnCheckName": { "type": "string" },
            "Active": { "type": "string" },
            "PrimaryPhone": {
              "type": "object",
              "properties": {
                "FreeFormNumber": { "type": "string" }
              }
            },
            "PrimaryEmailAddr": {
              "type": "object",
              "properties": {
                "Address": { "type": "string" }
              }
            },
            "EmployeeType": { "type": "string" },
            "status": { "type": "string" },
            "Id": { "type": "string" },
            "SyncToken": { "type": "string" },
            "MetaData": {
              "type": "object",
              "properties": {
                "CreateTime": { "type": "string" },
                "LastUpdatedTime": { "type": "string" }
              }
            },
            "PrimaryAddr": {
              "type": "object",
              "properties": {
                "Line1": { "type": "string" },
                "City": { "type": "string" },
                "CountrySubDivisionCode": { "type": "string" },
                "PostalCode": { "type": "string" }
              }
            }
          }
        }
      }
    },
    "time": { "type": "string" }
  }
}
```

Input mapping template (employee record example)

The following is the input mapping template that corresponds to the original JSON data for the employee record example:

```
#set($inputRoot = $input.path('$'))
{
  "QueryResponse": {
    "maxResults": "$inputRoot.QueryResponse.maxResults",
    "startPosition": "$inputRoot.QueryResponse.startPosition",
    "Employee": {
      "Organization": "$inputRoot.QueryResponse.Employee.Organization",
      "Title": "$inputRoot.QueryResponse.Employee.Title",
      "GivenName": "$inputRoot.QueryResponse.Employee.GivenName",
      "MiddleName": "$inputRoot.QueryResponse.Employee.MiddleName",
      "FamilyName": "$inputRoot.QueryResponse.Employee.FamilyName",
      "DisplayName": "$inputRoot.QueryResponse.Employee.DisplayName",
      "PrintOnCheckName": "$inputRoot.QueryResponse.Employee.PrintOnCheckName",
      "Active": "$inputRoot.QueryResponse.Employee.Active",
      "PrimaryPhone": { "FreeFormNumber": 
 "$inputRoot.QueryResponse.Employee.PrimaryPhone.FreeFormNumber" },
      "PrimaryEmailAddr": { "Address": 
 "$inputRoot.QueryResponse.Employee.PrimaryEmailAddr.Address" },
      "EmployeeType": "$inputRoot.QueryResponse.Employee.EmployeeType",
      "status": "$inputRoot.QueryResponse.Employee.status",
      "Id": "$inputRoot.QueryResponse.Employee.Id",
      "SyncToken": "$inputRoot.QueryResponse.Employee.SyncToken",
      "MetaData": {
        "CreateTime": "$inputRoot.QueryResponse.Employee.MetaData.CreateTime",
        "LastUpdatedTime": "$inputRoot.QueryResponse.Employee.MetaData.LastUpdatedTime"
      },
      "PrimaryAddr" : {
        "Line1": "$inputRoot.QueryResponse.Employee.PrimaryAddr.Line1",
        "City": "$inputRoot.QueryResponse.Employee.PrimaryAddr.City",
        "CountrySubDivisionCode": 
 "$inputRoot.QueryResponse.Employee.PrimaryAddr.CountrySubDivisionCode",
        "PostalCode": "$inputRoot.QueryResponse.Employee.PrimaryAddr.PostalCode"
      }
    }
  },
  "time": "$inputRoot.time"
}
```

Transformed data (employee record example)

The following is one example of how the original employee record example JSON data could be transformed for output:

```
{
  "QueryResponse": {
    "maxResults": "1",
    "startPosition": "1",
    "Employees": [
      {
        "Title": "Mrs.",
        "GivenName": "Jane",
        "MiddleName": "Lane",
        "FamilyName": "Doe",
        "DisplayName": "Jane Lane Doe",
        "PrintOnCheckName": "Jane Lane Doe",
        "Active": "true",
        "PrimaryPhone": "505.555.9999",
        "Email": [
          {
            "type": "primary",
            "Address": "janedoe@example.com"
          }
        ],
        "EmployeeType": "Regular",
        "PrimaryAddr": {
          "Line1": "123 Any Street",
          "City": "Any City",
          "CountrySubDivisionCode": "WA",
          "PostalCode": "01234"
        }
      }
    ]
  },
  "time": "2015-04-27T22:12:32.012Z"
}
```

Output model (employee record example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "EmployeeOutputModel",
  "type": "object",
  "properties": {
    "QueryResponse": {
      "type": "object",
      "properties": {
        "maxResults": { "type": "string" },
        "startPosition": { "type": "string" },
        "Employees": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "Title": { "type": "string" },
              "GivenName": { "type": "string" },
              "MiddleName": { "type": "string" },
              "FamilyName": { "type": "string" },
              "DisplayName": { "type": "string" },
              "PrintOnCheckName": { "type": "string" },
              "Active": { "type": "string" },
              "PrimaryPhone": { "type": "string" },
              "Email": {
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "type": { "type": "string" },
                    "Address": { "type": "string" }
                  }
                }
              },
              "EmployeeType": { "type": "string" },
              "PrimaryAddr": {
                "type": "object",
                "properties": {
                  "Line1": {"type": "string" },
                  "City": { "type": "string" },
                  "CountrySubDivisionCode": { "type": "string" },
                  "PostalCode": { "type": "string" }
                }
              }
            }
          }
        }
      }
    },
    "time": { "type": "string" }
  }
}
```

Output mapping template (employee record example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "QueryResponse": {
    "maxResults": "$inputRoot.QueryResponse.maxResults",
    "startPosition": "$inputRoot.QueryResponse.startPosition",
    "Employees": [
      {
        "Title": "$inputRoot.QueryResponse.Employee.Title",
        "GivenName": "$inputRoot.QueryResponse.Employee.GivenName",
        "MiddleName": "$inputRoot.QueryResponse.Employee.MiddleName",
        "FamilyName": "$inputRoot.QueryResponse.Employee.FamilyName",
        "DisplayName": "$inputRoot.QueryResponse.Employee.DisplayName",
        "PrintOnCheckName": "$inputRoot.QueryResponse.Employee.PrintOnCheckName",
        "Active": "$inputRoot.QueryResponse.Employee.Active",
        "PrimaryPhone":
 "$inputRoot.QueryResponse.Employee.PrimaryPhone.FreeFormNumber",
        "Email" : [
          {
            "type": "primary",
            "Address": "$inputRoot.QueryResponse.Employee.PrimaryEmailAddr.Address"
          }
        ],
        "EmployeeType": "$inputRoot.QueryResponse.Employee.EmployeeType",
        "PrimaryAddr": {
          "Line1": "$inputRoot.QueryResponse.Employee.PrimaryAddr.Line1",
          "City": "$inputRoot.QueryResponse.Employee.PrimaryAddr.City",
          "CountrySubDivisionCode":
 "$inputRoot.QueryResponse.Employee.PrimaryAddr.CountrySubDivisionCode",
          "PostalCode": "$inputRoot.QueryResponse.Employee.PrimaryAddr.PostalCode"
        }
      }
    ]
  },
  "time": "$inputRoot.time"
}
```

Amazon API Gateway API request and response data mapping reference

This section explains how to set up data mappings from an API's method request data, including other data stored in `context`, `stage`, or `util` variables, to the corresponding integration request parameters and from an integration response data, including the other data, to the method response parameters. The method request data includes request parameters (path, query string, headers) and the body. The integration response data includes response parameters (headers) and the body. For more information about using the stage variables, see Amazon API Gateway stage variables reference.

Topics

- Map method request data to integration request parameters
- Map integration response data to method response headers
- Map request and response payloads between method and integration
- Integration passthrough behaviors

Map method request data to integration request parameters

Integration request parameters, in the form of path variables, query strings or headers, can be mapped from any defined method request parameters and the payload.

In the following table, *`PARAM_NAME`* is the name of a method request parameter of the given parameter type. It must match the regular expression `'^[a-zA-Z0-9._$-]+$]'`. It must have been defined before it can be referenced. *`JSONPath_EXPRESSION`* is a JSONPath expression for a JSON field of the body of a request or response.

> **Note**
>
> The "$" prefix is omitted in this syntax.

Integration request data mapping expressions

Mapped data source	Mapping expression
Method request path	`method.request.path. PARAM_NAME`
Method request query string	`method.request.querystring. PARAM_NAME`
Multi-value method request query string	`method.request.multivaluequerystring. PARAM_NAME`
Method request header	`method.request.header. PARAM_NAME`
Multi-value method request header	`method.request.multivalueheader. PARAM_NAME`
Method request body	`method.request.body`
Method request body (JsonPath)	`method.request.body. JSONPath_EXPRESSION .`
Stage variables	`stageVariables. VARIABLE_NAME`
Context variables	`context.VARIABLE_NAME` that must be one of the supported context variables.
Static value	`'STATIC_VALUE'` . The `STATIC_VALUE` is a string literal and must be enclosed within a pair of single quotes.

Example Mappings from method request parameter in OpenAPI

The following example shows an OpenAPI snippet that maps:

- the method request's header, named `methodRequestHeaderParam`, into the integration request path parameter, named `integrationPathParam`
- the multi-value method request query string, named `methodRequestQueryParam`, into the integration request query string, named `integrationQueryParam`

```
...
"requestParameters" : {

    "integration.request.path.integrationPathParam" :
 "method.request.header.methodRequestHeaderParam",
    "integration.request.querystring.integrationQueryParam" :
 "method.request.multivaluequerystring.methodRequestQueryParam"

}
...
```

Integration request parameters can also be mapped from fields in the JSON request body using a JSONPath expression. The following table shows the mapping expressions for a method request body and its JSON fields.

Example Mapping from method request body in OpenAPI

The following example shows an OpenAPI snippet that maps 1) the method request body to the integration request header, named `body-header`, and 2) a JSON field of the body, as expressed by a JSON expression (`petstore.pets[0].name`, without the `$.` prefix).

```
...
"requestParameters" : {

    "integration.request.header.body-header" : "method.request.body",
    "integration.request.path.pet-name" : "method.request.body.petstore.pets[0].name",

}
...
```

Map integration response data to method response headers

Method response header parameters can be mapped from any integration response header or integration response body, `$context` variables, or static values.

Method response header mapping expressions

Mapped data source	Mapping expression
Integration response header	`integration.response.header.` *`PARAM_NAME`*
Integration response header	`integration.response.multivalueheader.` *`PARAM_NAME`*
Integration response body	`integration.response.body`
Integration response body (JsonPath)	`integration.response.body.` *`JSONPath_EXPRESSION`*
Stage variable	`stageVariables.` *`VARIABLE_NAME`*
Context variable	`context.`*`VARIABLE_NAME`* that must be one of the supported context variables.
Static value	*`'STATIC_VALUE'`* . The *`STATIC_VALUE`* is a string literal and must be enclosed within a pair of single quotes.

Example Data mapping from integration response in OpenAPI

The following example shows an OpenAPI snippet that maps 1) the integration response's `redirect.url`, JSONPath field into the request response's `location` header; and 2) the integration response's `x-app-id` header to the method response's `id` header.

```

...
"responseParameters" : {

    "method.response.header.location" : "integration.response.body.redirect.url",
    "method.response.header.id" : "integration.response.header.x-app-id",
    "method.response.header.items" : "integration.response.multivalueheader.item",

}
...
```

Map request and response payloads between method and integration

API Gateway uses Velocity Template Language (VTL) engine to process body mapping templates for the integration request and integration response. The mapping templates translate method request payloads to the corresponding integration request payloads and translate integration response bodies to the method response bodies.

The VTL templates use JSONPath expressions, other parameters such as calling contexts and stage variables, and utility functions to process the JSON data.

If a model is defined to describe the data structure of a payload, API Gateway can use the model to generate a skeletal mapping template for an integration request or integration response. You can use the skeletal template as an aid to customize and expand the mapping VTL script. However, you can create a mapping template from scratch without defining a model for the payload's data structure.

Select a VTL mapping template

API Gateway uses the following logic to select a mapping template, in Velocity Template Language (VTL), to map the payload from a method request to the corresponding integration request or to map the payload from an integration response to the corresponding method response.

For a request payload, API Gateway uses the request’s `Content-Type` header value as the key to select the mapping template for the request payload. For a response payload, API Gateway uses the incoming request’s `Accept` header value as the key to select the mapping template.

When the `Content-Type` header is absent in the request, API Gateway assumes that its default value is `application/json`. For such a request, API Gateway uses `application/json` as the default key to select the mapping template, if one is defined. When no template matches this key, API Gateway passes the payload through unmapped if the passthroughBehavior property is set to `WHEN_NO_MATCH` or `WHEN_NO_TEMPLATES`.

When the `Accept` header is not specified in the request, API Gateway assumes that its default value is `application/json`. In this case, API Gateway selects an existing mapping template for `application/json` to map the response payload. If no template is defined for `application/json`, API Gateway selects the first existing template and uses it as the default to map the response payload. Similarly, API Gateway uses the first existing template when the specified

Accept header value does not match any existing template key. If no template is defined, API Gateway simply passes the response payload through unmapped.

For example, suppose that an API has a application/json template defined for a request payload and has a application/xml template defined for the response payload. If the client sets the "Content-Type : application/json", and "Accept : application/xml" headers in the request, both the request and response payloads will be processed with the corresponding mapping templates. If the Accept:application/xml header is absent, the application/xml mapping template will be used to map the response payload. To return the response payload unmapped instead, you must set up an empty template for application/json.

Only the MIME type is used from the Accept and Content-Type headers when selecting a mapping template. For example, a header of "Content-Type: application/json; charset=UTF-8" will have a request template with the application/json key selected.

Integration passthrough behaviors

With non-proxy integrations, when a method request carries a payload and either the Content-Type header does not match any specified mapping template or no mapping template is defined, you can choose to pass the client-supplied request payload through the integration request to the backend without transformation. The process is known as integration passthrough.

For proxy integrations, API Gateway passes the entire request through to your backend, and you do not have the option to modify the passthrough behaviors.

The actual passthrough behavior of an incoming request is determined by the option you choose for a specified mapping template, during integration request set-up, and the Content Type header that a client set in the incoming request. There are three options:

When no template matches the request Content-Type header

Choose this option if you want the method request body to pass through the integration request to the backend without transformation when the method request content type does not match any content types associated with the mapping templates.

When calling the API Gateway API, you choose this option by setting WHEN_NO_MATCH as the passthroughBehavior property value on the Integration.

When there are no templates defined (recommended)

Choose this option if you want the method request body to pass through the integration request to the backend without transformation when no mapping template is defined in the integration request. If a template is defined when this option is selected, the method request of an unmapped content type will be rejected with an HTTP 415 Unsupported Media Type response.

When calling the API Gateway API, you choose this option by setting `WHEN_NO_TEMPLATES` as the `passthroughBehavior` property value on the Integration.

Never

Choose this option if you do not want the method request body to pass through the integration request to the backend without transformation when no mapping template is defined in the integration request. If a template is defined when this option is selected, the method request of an unmapped content type will be rejected with an HTTP 415 Unsupported Media Type response.

When calling the API Gateway API, you choose this option by setting `NEVER` as the `passthroughBehavior` property value on the Integration.

The following examples illustrate the possible passthrough behaviors.

Example 1: One mapping template is defined in the integration request for the `application/json` content type.

Content-type header \Selected passthrough option	`WHEN_NO_MATCH`	`WHEN_NO_TEMPLATES`	`NEVER`
None (default to `application/json`)	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
`application/json`	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
`application/xml`	The request payload is not transformed and is sent to the backend as-is.	The request is rejected with an HTTP 415 `Unsupported`	The request is rejected with an HTTP 415 `Unsupported`

Content-type header \Selected passthrough option	**`WHEN_NO_MATCH`**	**`WHEN_NO_TEMPLATES`**	**`NEVER`**
		`Media Type` response.	`Media Type` response.

Example 2: One mapping template is defined in the integration request for the `application/xml` content type.

Content-type header \Selected passthrough option	**`WHEN_NO_MATCH`**	**`WHEN_NO_TEMPLATES`**	**`NEVER`**
None (default to `application/json`)	The request payload is not transformed and is sent to the backend as-is.	The request is rejected with an HTTP 415 `Unsupported Media Type` response.	The request is rejected with an HTTP 415 `Unsupported Media Type` response.
`application/json`	The request payload is not transformed and is sent to the backend as-is.	The request is rejected with an HTTP 415 `Unsupported Media Type` response.	The request is rejected with an HTTP 415 `Unsupported Media Type` response.
`application/xml`	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.

API Gateway mapping template and access logging variable reference

This section provides reference information for the variables and functions that Amazon API Gateway defines for use with data models, authorizers, mapping templates, and CloudWatch access logging. For detailed information about how to use these variables and functions, see Understanding mapping templates. For more information about the Velocity Template Language (VTL), see the VTL Reference.

Topics

- $context Variables for data models, authorizers, mapping templates, and CloudWatch access logging
- $context Variable template example
- $context Variables for access logging only
- $input Variables
- $input Variable template examples
- $stageVariables
- $util Variables

> **Note**
>
> For `$method` and `$integration` variables, see the section called “Request and response data mapping reference”.

`$context` Variables for data models, authorizers, mapping templates, and CloudWatch access logging

The following `$context` variables can be used in data models, authorizers, mapping templates, and CloudWatch access logging.

For `$context` variables that can be used only in CloudWatch access logging, see the section called “`$context` Variables for access logging only”.

Parameter	Description
`$context.accountId`	The API owner's AWS account ID.
`$context.apiId`	The identifier API Gateway assigns to your API.
`$context.authorizer.claims.` *`property`*	A property of the claims returned from the Amazon Cognito user pool after the method

Parameter	Description
	caller is successfully authenticated. For more information, see the section called "Use Amazon Cognito user pool as authorizer for a REST API". **Note** Calling `$context.authorizer.claims` returns null.
`$context.authorizer.principalId`	The principal user identification associated with the token sent by the client and returned from an API Gateway Lambda authorizer (formerly known as a custom authorizer). For more information, see the section called "Use Lambda authorizers".

Parameter	Description
`$context.authorizer. property`	The stringified value of the specified key-value pair of the `context` map returned from an API Gateway Lambda authorizer function. For example, if the authorizer returns the following `context` map: `"context" : { "key": "value", "numKey": 1, "boolKey": true }` calling `$context.authorizer.key` returns the `"value"` string, calling `$context.authorizer.numKey` returns the `"1"` string, and calling `$context.authorizer.boolKey` returns the `"true"` string. For more information, see the section called "Use Lambda authorizers".
`$context.awsEndpointRequestId`	The AWS endpoint's request ID.
`$context.domainName`	The full domain name used to invoke the API. This should be the same as the incoming `Host` header.
`$context.domainPrefix`	The first label of the `$context.domainName`.

Parameter	Description
`$context.error.message`	A string containing an API Gateway error message. This variable can only be used for simple variable substitution in a GatewayResponse body-mapping template, which is not processed by the Velocity Template Language engine, and in access logging. For more information, see the section called "Metrics" and the section called "Setting up gateway responses to customize error responses".
`$context.error.messageString`	The quoted value of `$context.error.message`, namely `"$context.error.message"`.
`$context.error.responseType`	A type of GatewayResponse. This variable can only be used for simple variable substitution in a GatewayResponse body-mapping template, which is not processed by the Velocity Template Language engine, and in access logging. For more information, see the section called "Metrics" and the section called "Setting up gateway responses to customize error responses".
`$context.error.validationErrorString`	A string containing a detailed validation error message.
`$context.extendedRequestId`	The extended ID that API Gateway generates and assigns to the API request. The extended request ID contains useful information for debugging and troubleshooting.
`$context.httpMethod`	The HTTP method used. Valid values include: `DELETE`, `GET`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, and `PUT`.

Parameter	Description
`$context.identity.accountId`	The AWS account ID associated with the request.
`$context.identity.apiKey`	For API methods that require an API key, this variable is the API key associated with the method request. For methods that don't require an API key, this variable is null. For more information, see the section called "Usage plans".
`$context.identity.apiKeyId`	The API key ID associated with an API request that requires an API key.
`$context.identity.caller`	The principal identifier of the caller that signed the request. Supported for resources that use IAM authorization.
`$context.identity.cognitoAuthenticationProvider`	A comma-separated list of the Amazon Cognito authentication providers used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For example, for an identity from an Amazon Cognito user pool, `cognito-idp.`*`region`*`.amazonaws.com/`*`user_pool_id`*`,cognito-idp.`*`region`*`.amazonaws.com/`*`user_pool_id`*`:CognitoSignIn:`*`token subject claim`* For information, see Using Federated Identities in the *Amazon Cognito Developer Guide*.

Parameter	Description
`$context.identity.cognitoAuthenticationType`	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials. Possible values include `authenticated` for authenticated identities and `unauthenticated` for unauthenticated identities.
`$context.identity.cognitoIdentityId`	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.cognitoIdentityPoolId`	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.principalOrgId`	The AWS organization ID.
`$context.identity.sourceIp`	The source IP address of the immediate TCP connection making the request to API Gateway endpoint.
`$context.identity.clientCert.clientCertPem`	The PEM-encoded client certificate that the client presented during mutual TLS authentication. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.

Parameter	Description
`$context.identity.clientCert.subjectDN`	The distinguished name of the subject of the certificate that a client presents. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.
`$context.identity.clientCert.issuerDN`	The distinguished name of the issuer of the certificate that a client presents. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.
`$context.identity.clientCert.serialNumber`	The serial number of the certificate. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.
`$context.identity.clientCert.validity.notBefore`	The date before which the certificate is invalid. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.
`$context.identity.clientCert.validity.notAfter`	The date after which the certificate is invalid. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled. Present only in access logs if mutual TLS authentication fails.
`$context.identity.user`	The principal identifier of the user that will be authorized against resource access. Supported for resources that use IAM authorization.

Parameter	Description
`$context.identity.userAgent`	The `User-Agent` header of the API caller.
`$context.identity.userArn`	The Amazon Resource Name (ARN) of the effective user identified after authentication. For more information, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html.
`$context.path`	The request path. For example, for a non-proxy request URL of `https://{rest-api-id}.execute-api.{region}.amazonaws.com/{stage}/root/child`, the `$context.path` value is `/{stage}/root/child`.
`$context.protocol`	The request protocol, for example, `HTTP/1.1`. **Note** API Gateway APIs can accept HTTP/2 requests, but API Gateway sends requests to backend integrations using HTTP/1.1. As a result, the request protocol is logged as HTTP/1.1 even if a client sends a request that uses HTTP/2.
`$context.requestId`	An ID for the request. Clients can override this request ID. Use `$context.extendedRequestId` for a unique request ID that API Gateway generates.

Parameter	Description
`$context.requestOverride.header. header_name`	The request header override. If this parameter is defined, it contains the headers to be used instead of the **HTTP Headers** that are defined in the **Integration Request** pane. For more information, see Use a mapping template to override an API's request and response parameters and status codes.
`$context.requestOverride.path. path_name`	The request path override. If this parameter is defined, it contains the request path to be used instead of the **URL Path Parameters** that are defined in the **Integration Request** pane. For more information, see Use a mapping template to override an API's request and response parameters and status codes.
`$context.requestOverride.querystring. querystring_name`	The request query string override. If this parameter is defined, it contains the request query strings to be used instead of the **URL Query String Parameters** that are defined in the **Integration Request** pane. For more information, see Use a mapping template to override an API's request and response parameters and status codes.
`$context.responseOverride.header. header_name`	The response header override. If this parameter is defined, it contains the header to be returned instead of the **Response header** that is defined as the **Default mapping** in the **Integration Response** pane. For more information, see Use a mapping template to override an API's request and response parameters and status codes.

Parameter	Description
`$context.responseOverride.status`	The response status code override. If this parameter is defined, it contains the status code to be returned instead of the **Method response status** that is defined as the **Default mapping** in the **Integration Response** pane. For more information, see Use a mapping template to override an API's request and response parameters and status codes.
`$context.requestTime`	The CLF-formatted request time (`dd/MMM/yyyy:HH:mm:ss +-hhmm`).
`$context.requestTimeEpoch`	The Epoch-formatted request time, in milliseconds.
`$context.resourceId`	The identifier that API Gateway assigns to your resource.
`$context.resourcePath`	The path to your resource. For example, for the non-proxy request URI of `https://{rest-api-id}.execute-api.{region}.amazonaws.com/{stage}/root/child` , The `$context.resourcePath` value is `/root/child` . For more information, see Tutorial: Build a REST API with HTTP non-proxy integration.
`$context.stage`	The deployment stage of the API request (for example, `Beta` or `Prod`).
`$context.wafResponseCode`	The response received from AWS WAF: `WAF_ALLOW` or `WAF_BLOCK` . Will not be set if the stage is not associated with a web ACL. For more information, see the section called "AWS WAF".

Parameter	Description
`$context.webaclArn`	The complete ARN of the web ACL that is used to decide whether to allow or block the request. Will not be set if the stage is not associated with a web ACL. For more information, see the section called "AWS WAF".

`$context` Variable template example

You might want to use `$context` variables in a mapping template if your API method passes structured data to a backend that requires the data to be in a particular format.

The following example shows a mapping template that maps incoming `$context` variables to backend variables with slightly different names in an integration request payload:

> **Note**
>
> One of the variables is an API key. This example assumes that the method requires an API key.

```
{
    "stage" : "$context.stage",
    "request_id" : "$context.requestId",
    "api_id" : "$context.apiId",
    "resource_path" : "$context.resourcePath",
    "resource_id" : "$context.resourceId",
    "http_method" : "$context.httpMethod",
    "source_ip" : "$context.identity.sourceIp",
    "user-agent" : "$context.identity.userAgent",
    "account_id" : "$context.identity.accountId",
    "api_key" : "$context.identity.apiKey",
    "caller" : "$context.identity.caller",
    "user" : "$context.identity.user",
    "user_arn" : "$context.identity.userArn"
}
```

The output of this mapping template should look like the following:

```
{
  stage: 'prod',
  request_id: 'abcdefg-000-000-0000-abcdefg',
  api_id: 'abcd1234',
  resource_path: '/',
  resource_id: 'efg567',
  http_method: 'GET',
  source_ip: '192.0.2.1',
  user-agent: 'curl/7.84.0',
  account_id: '111122223333',
  api_key: 'MyTestKey',
  caller: 'ABCD-0000-12345',
  user: 'ABCD-0000-12345',
  user_arn: 'arn:aws:sts::111122223333:assumed-role/Admin/carlos-salazar'
}
```

`$context` Variables for access logging only

The following `$context` variables are available only for access logging. For more information, see the section called "CloudWatch logs". (For WebSocket APIs, see the section called "Metrics".)

Parameter	Description
`$context.authorize.error`	The authorization error message.
`$context.authorize.latency`	The authorization latency in ms.
`$context.authorize.status`	The status code returned from an authorization attempt.
`$context.authorizer.error`	The error message returned from an authorizer.
`$context.authorizer.integrationLatency`	The authorizer latency in ms.
`$context.authorizer.integrationStatus`	The status code returned from a Lambda authorizer.
`$context.authorizer.latency`	The authorizer latency in ms.

Parameter	Description
`$context.authorizer.requestId`	The AWS endpoint's request ID.
`$context.authorizer.status`	The status code returned from an authorizer.
`$context.authenticate.error`	The error message returned from an authentication attempt.
`$context.authenticate.latency`	The authentication latency in ms.
`$context.authenticate.status`	The status code returned from an authentication attempt.
`$context.customDomain.basePathMatched`	The path for an API mapping that an incoming request matched. Applicable when a client uses a custom domain name to access an API. For example if a client sends a request to `https://api.example.com/v1/orders/1234`, and the request matches the API mapping with the path `v1/orders`, the value is `v1/orders`. To learn more, see the section called "API mappings".
`$context.integration.error`	The error message returned from an integration.
`$context.integration.integrationStatus`	For Lambda proxy integration, the status code returned from AWS Lambda, not from the backend Lambda function code.
`$context.integration.latency`	The integration latency in ms. Equivalent to `$context.integrationLatency`.
`$context.integration.requestId`	The AWS endpoint's request ID. Equivalent to `$context.awsEndpointRequestId`.

Parameter	Description
`$context.integration.status`	The status code returned from an integrati on. For Lambda proxy integrations, this is the status code that your Lambda function code returns.
`$context.integrationLatency`	The integration latency in ms.
`$context.integrationStatus`	For Lambda proxy integration, this parameter represents the status code returned from AWS Lambda, not from the backend Lambda function code.
`$context.responseLatency`	The response latency in ms.
`$context.responseLength`	The response payload length in bytes.
`$context.status`	The method response status.
`$context.waf.error`	The error message returned from AWS WAF.
`$context.waf.latency`	The AWS WAF latency in ms.
`$context.waf.status`	The status code returned from AWS WAF.
`$context.xrayTraceId`	The trace ID for the X-Ray trace. For more information, see the section called "Setting up AWS X-Ray".

`$input` Variables

The `$input` variable represents the method request payload and parameters to be processed by a mapping template. It provides the following functions:

Variable and function	Description
`$input.body`	Returns the raw request payload as a string.

Variable and function	Description
`$input.json(x)`	This function evaluates a JSONPath expression and returns the results as a JSON string. For example, `$input.json('$.pets')` returns a JSON string representing the `pets` structure. For more information about JSONPath, see JSONPath or JSONPath for Java.
`$input.params()`	Returns a map of all the request parameters. We recommend that you use `$util.escapeJavaScript` to sanitize the result to avoid a potential injection attack. For full control of request sanitization, use a proxy integration without a template and handle request sanitization in your integration.
`$input.params(x)`	Returns the value of a method request parameter from the path, query string, or header value (searched in that order), given a parameter name string `x`. We recommend that you use `$util.escapeJavaScript` to sanitize the parameter to avoid a potential injection attack. For full control of parameter sanitization, use a proxy integration without a template and handle request sanitization in your integration.

<table>
<tr><th>Variable and function</th><th>Description</th></tr>
<tr><td><code>$input.path(x)</code></td><td>
<p>Takes a JSONPath expression string (x) and returns a JSON object representation of the result. This allows you to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL).</p>
<p>For example, if the expression <code>$input.path('$.pets')</code> returns an object like this:</p>
<pre>
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
</pre>
<p><code>$input.path('$.pets').count()</code> would return <code>"3"</code>.</p>
<p>For more information about JSONPath, see JSONPath or JSONPath for Java.</p>
</td></tr>
</table>

$input Variable template examples

The following examples show how to use the `$input` variables in mapping templates. You can use a mock integration or a Lambda non-proxy integration that returns the input event back to API Gateway to try these examples.

Parameter mapping template example

The following example passes all request parameters, including `path`, `querystring`, and `header`, through to the integration endpoint via a JSON payload:

```
#set($allParams = $input.params())
{
  "params" : {
    #foreach($type in $allParams.keySet())
    #set($params = $allParams.get($type))
    "$type" : {
      #foreach($paramName in $params.keySet())
      "$paramName" : "$util.escapeJavaScript($params.get($paramName))"
      #if($foreach.hasNext),#end
      #end
    }
    #if($foreach.hasNext),#end
    #end
  }
}
```

For a request that includes the following input parameters:

- A path parameter named `myparam`
- Query string parameters `querystring1=value1,value2&querystring2=value3`
- Headers `"header1" : "value1"`, `"header2" : "value2"`, `"header3" : "value3"`.

The output of this mapping template should look like the following:

```
{
  "params" : {
        "path" : {
            "path" : "myparam"
                }
```

```
    ,        "querystring" : {
            "querystring1" : "value1,value2"
      ,            "querystring2" : "value3"
                }
    ,        "header" : {
            "header3" : "value3"
      ,            "header2" : "value2"
      ,            "header1" : "value1"
                }
          }
}
```

JSON mapping template example

You might want to use the `$input` variable to get query strings and the request body with or without using models. You might also want to get the parameter and the payload, or a subsection of the payload. The following three examples show how to do this.

The following example uses a mapping template to get a subsection of the payload. This example get the input parameter `name` and then the the entire POST body:

```
{
    "name" : "$input.params('name')",
    "body" : $input.json('$')
}
```

For a request that includes the query string parameters `name=Bella&type=dog` and the following body:

```
{
    "Price" : "249.99",
    "Age": "6"
}
```

The output of this mapping template should look like the following:

```
{
    "name" : "Bella",
    "body" : {"Price":"249.99","Age":"6"}
}
```

If the JSON input contains unescaped characters that cannot be parsed by JavaScript, API Gateway might return a 400 response. Apply `$util.escapeJavaScript($input.json('$'))` to ensure the JSON input can be parsed properly.

The previous example with `$util.escapeJavaScript($input.json('$'))` applied is as follows:

```
{
    "name" : "$input.params('name')",
    "body" : $util.escapeJavaScript($input.json('$'))
}
```

In this case, the output of this mapping template should look like the following:

```
{
    "name" : "Bella",
    "body": {\"Price\":\"249.99\",\"Age\":\"6\"}
}
```

JSONPath expression example

The following example shows how to pass a JSONPath expression to the `json()` method. You could also read a subsection of your request body object by using a period, `.`, to specify a property:

```
{
    "name" : "$input.params('name')",
    "body" : $input.json('$.Age')
}
```

For a request that includes the query string parameters `name=Bella&type=dog` and the following body:

```
{
    "Price" : "249.99",
    "Age": "6"
}
```

The output of this mapping template should look like the following:

```
{
    "name" : "Bella",
    "body" : "6"
}
```

If a method request payload contains unescaped characters that cannot be parsed by JavaScript, API Gateway might return a `400` response. Apply `$util.escapeJavaScript()` to ensure the JSON input can be parsed properly.

The previous example with `$util.escapeJavaScript($input.json('$.Age'))` applied is as follows:

```
{
    "name" : "$input.params('name')",
    "body" : "$util.escapeJavaScript($input.json('$.Age'))"
}
```

In this case, the output of this mapping template should look like the following:

```
{
    "name" : "Bella",
    "body": "\"6\""
}
```

Request and response example

The following example uses `$input.params()`, `$input.path()`, and `$input.json()` for a resource with the path `/things/{id}`:

```
{
    "id" : "$input.params('id')",
    "count" : "$input.path('$.things').size()",
    "things" : $input.json('$.things')"
}
```

For a request that includes the path parameter `123` and the following body:

```
{
      "things": {
            "1": {},
            "2": {},
            "3": {}
      }
}
```

The output of this mapping template should look like the following:

```
{"id":"123","count":"3","things":{"1":{},"2":{},"3":{}}}
```

If a method request payload contains unescaped characters that cannot be parsed by JavaScript, API Gateway might return a `400` response. Apply `$util.escapeJavaScript()` to ensure the JSON input can be parsed properly.

The previous example with `$util.escapeJavaScript($input.json('$.things'))` applied is as follows:

```
{
     "id" : "$input.params('id')",
     "count" : "$input.path('$.things').size()",
     "things" : "$util.escapeJavaScript($input.json('$.things'))"
}
```

The output of this mapping template should look like the following:

```
{"id":"123","count":"3","things":"{\"1\":{},\"2\":{},\"3\":{}}"}
```

For more mapping examples, see Understanding mapping templates.

`$stageVariables`

Stage variables can be used in parameter mapping and mapping templates and as placeholders in ARNs and URLs used in method integrations. For more information, see the section called "Set up stage variables".

Syntax	Description
`$stageVariables.` *`<variable_name>`* `,` `$stageVariables['` *`<variable_name>`* `']`, or `${stageVariables['` *`<variable_name>`* `']}`	*`<variable_name>`* represents a stage variable name.

$util Variables

The `$util` variable contains utility functions for use in mapping templates.

> **Note**
>
> Unless otherwise specified, the default character set is UTF-8.

Function	Description
`$util.escapeJavaScript()`	Escapes the characters in a string using JavaScript string rules. **Note** This function will turn any regular single quotes (`'`) into escaped ones (`\'`). However, the escaped single quotes are not valid in JSON. Thus, when the output from this function is used in a JSON property, you must turn any escaped single quotes (`\'`) back to regular single quotes (`'`). This is shown in the following example: `"input" : "$util.escapeJavaScript( data).replaceAll("\\'","'")"`
`$util.parseJson()`	Takes "stringified" JSON and returns an object representation of the result. You can use the result from this function to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL). For example, if you have the following payload:

Function	Description

	`{"errorMessage":"{\"key1\":\"var1\",\"key2\":{\"arr\":[1,2,3]}}"}`
	and use the following mapping template
	`#set ($errorMessageObj = $util.parseJson($input.path('$.errorMessage')))`
	`{`
	`   "errorMessageObjKey2ArrVal" : $errorMessageObj.key2.arr[0]`
	`}`
	You will get the following output:
	`{`
	`   "errorMessageObjKey2ArrVal" : 1`
	`}`

`$util.urlEncode()`	Converts a string into "application/x-www-form-urlencoded" format.
`$util.urlDecode()`	Decodes an "application/x-www-form-urlencoded" string.
`$util.base64Encode()`	Encodes the data into a base64-encoded string.
`$util.base64Decode()`	Decodes the data from a base64-encoded string.

Gateway responses in API Gateway

A gateway response is identified by a response type that is defined by API Gateway. The response consists of an HTTP status code, a set of additional headers that are specified by parameter mappings, and a payload that is generated by a non-VTL mapping template.

In the API Gateway REST API, a gateway response is represented by the GatewayResponse. In OpenAPI, a `GatewayResponse` instance is described by the x-amazon-apigateway-gateway-responses.gatewayResponse extension.

To enable a gateway response, you set up a gateway response for a supported response type at the API level. Whenever API Gateway returns a response of this type, the header mappings and payload mapping templates defined in the gateway response are applied to return the mapped results to the API caller.

In the following section, we show how to set up gateway responses by using the API Gateway console and the API Gateway REST API.

Setting up gateway responses to customize error responses

If API Gateway fails to process an incoming request, it returns to the client an error response without forwarding the request to the integration backend. By default, the error response contains a short descriptive error message. For example, if you attempt to call an operation on an undefined API resource, you receive an error response with the `{ "message": "Missing Authentication Token" }` message. If you are new to API Gateway, you may find it difficult to understand what actually went wrong.

For some of the error responses, API Gateway allows customization by API developers to return the responses in different formats. For the `Missing Authentication Token` example, you can add a hint to the original response payload with the possible cause, as in this example: `{"message":"Missing Authentication Token", "hint":"The HTTP method or resources may not be supported."}`.

When your API mediates between an external exchange and the AWS Cloud, you use VTL mapping templates for integration request or integration response to map the payload from one format to another. However, the VTL mapping templates work only for valid requests with successful responses.

For invalid requests, API Gateway bypasses the integration altogether and returns an error response. You must use the customization to render the error responses in an exchange-compliant format. Here, the customization is rendered in a non-VTL mapping template supporting only simple variable substitutions.

Generalizing the API Gateway-generated error response to any responses generated by API Gateway, we refer to them as *gateway responses*. This distinguishes API Gateway-generated responses from the integration responses. A gateway response mapping template can access `$context` variable values and `$stageVariables` property values, as well as method request parameters, in the form of `method.request.*param-position*.*param-name*`.

For more information about `$context` variables, see [$context Variables for data models, authorizers, mapping templates, and CloudWatch access logging]. For more information about `$stageVariables`, see [$stageVariables]. For more information about method request parameters, see [the section called "$input Variables"].

Topics

- Set up a gateway response for a REST API using the API Gateway console
- Set up a gateway response using the API Gateway REST API
- Set up gateway response customization in OpenAPI
- Gateway response types

Set up a gateway response for a REST API using the API Gateway console

To customize a gateway response using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **Gateway responses**.
4. Choose a response type, and then choose **Edit**. In this walkthrough, we use **Missing authentication token** as an example.
5. You can change the API Gateway-generated **Status code** to return a different status code that meets your API's requirements. In this example, the customization changes the status code from the default (`403`) to `404` because this error message occurs when a client calls an unsupported or invalid resource that can be thought of as not found.
6. To return custom headers, choose **Add response header** under **Response headers**. For illustration purposes, we add the following custom headers:

```
Access-Control-Allow-Origin:'a.b.c'
x-request-id:method.request.header.x-amzn-RequestId
x-request-path:method.request.path.petId
x-request-query:method.request.querystring.q
```

In the preceding header mappings, a static domain name (`'a.b.c'`) is mapped to the `Allow-Control-Allow-Origin` header to allow CORS access to the API; the input request header of `x-amzn-RequestId` is mapped to `request-id` in the response; the `petId` path variable of the incoming request is mapped to the `request-path` header in the response; and the `q` query parameter of the original request is mapped to the `request-query` header of the response.

7. Under **Response templates**, keep `application/json` for **Content Type** and enter the following body mapping template in the **Template body** editor:

```
{
     "message":"$context.error.messageString",
     "type": "$context.error.responseType",
     "statusCode": "'404'",
     "stage": "$context.stage",
     "resourcePath": "$context.resourcePath",
     "stageVariables.a": "$stageVariables.a"
}
```

   This example shows how to map `$context` and `$stageVariables` properties to properties of the gateway response body.

8. Choose **Save changes**.
9. Deploy the API to a new or existing stage.

Test your gateway response by calling the following CURL command, assuming the corresponding API method's invoke URL is `https://`*`o81lxisefl`*`.execute-api.us-east-1.amazonaws.com/custErr/pets/{petId}`:

```
curl -v -H 'x-amzn-RequestId:123344566' https://o81lxisefl.execute-api.us-
east-1.amazonaws.com/custErr/pets/5/type?q=1
```

Because the extra query string parameter q=1 isn't compatible with the API, an error is returned to trigger the specified gateway response. You should get a gateway response similar to the following:

```
> GET /custErr/pets/5?q=1 HTTP/1.1
Host: o81lxisefl.execute-api.us-east-1.amazonaws.com
User-Agent: curl/7.51.0
Accept: */*
```

```
HTTP/1.1 404 Not Found
Content-Type: application/json
Content-Length: 334
Connection: keep-alive
Date: Tue, 02 May 2017 03:15:47 GMT
x-amzn-RequestId: 123344566
Access-Control-Allow-Origin: a.b.c
x-amzn-ErrorType: MissingAuthenticationTokenException
header-1: static
x-request-query: 1
x-request-path: 5
X-Cache: Error from cloudfront
Via: 1.1 441811a054e8d055b893175754efd0c3.cloudfront.net (CloudFront)
X-Amz-Cf-Id: nNDR-fX4csbRoAgtQJ16u0rTDz9FZWT-Mk93KgoxnfzDlTUh3flmzA==

{
     "message":"Missing Authentication Token",
     "type": MISSING_AUTHENTICATION_TOKEN,
     "statusCode": '404',
     "stage": custErr,
     "resourcePath": /pets/{petId},
     "stageVariables.a": a
}
```

The preceding example assumes that the API backend is Pet Store and the API has a stage variable, a, defined.

Set up a gateway response using the API Gateway REST API

Before customizing a gateway response using the API Gateway REST API, you must have already created an API and have obtained its identifier. To retrieve the API identifier, you can follow restapi:gateway-responses link relation and examine the result.

To customize a gateway response using the API Gateway REST API

1. To overwrite an entire GatewayResponse instance, call the gatewayresponse:put action. Specify a desired responseType in the URL path parameter, and supply in the request payload the statusCode, responseParameters, and responseTemplates mappings.
2. To update part of a `GatewayResponse` instance, call the gatewayresponse:update action. Specify a desired `responseType` in the URL path parameter, and supply in the

request payload the individual `GatewayResponse` properties you want—for example, the `responseParameters` or the `responseTemplates` mapping.

Set up gateway response customization in OpenAPI

You can use the `x-amazon-apigateway-gateway-responses` extension at the API root level to customize gateway responses in OpenAPI. The following OpenAPI definition shows an example for customizing the GatewayResponse of the `MISSING_AUTHENTICATION_TOKEN` type.

```
  "x-amazon-apigateway-gateway-responses": {
    "MISSING_AUTHENTICATION_TOKEN": {
      "statusCode": 404,
      "responseParameters": {
        "gatewayresponse.header.x-request-path": "method.input.params.petId",
        "gatewayresponse.header.x-request-query": "method.input.params.q",
        "gatewayresponse.header.Access-Control-Allow-Origin": "'a.b.c'",
        "gatewayresponse.header.x-request-header": "method.input.params.Accept"
      },
      "responseTemplates": {
        "application/json": "{\n     \"message\": $context.error.messageString,\n
    \"type\":  \"$context.error.responseType\",\n     \"stage\":  \"$context.stage
\",\n     \"resourcePath\":  \"$context.resourcePath\",\n     \"stageVariables.a\":
 \"$stageVariables.a\",\n     \"statusCode\": \"'404'\"\n}"
      }
    }
```

In this example, the customization changes the status code from the default (`403`) to `404`. It also adds to the gateway response four header parameters and one body mapping template for the `application/json` media type.

Gateway response types

API Gateway exposes the following gateway responses for customization by API developers.

Gateway response type	Default status code	Description
ACCESS_DENIED	403	The gateway response for authorization failure—f or example, when access is denied by a custom or

Gateway response type	Default status code	Description
		Amazon Cognito authorizer. If the response type is unspecified, this response defaults to the DEFAULT_4XX type.
API_CONFIGURATION_ERROR	500	The gateway response for an invalid API configuration—including when an invalid endpoint address is submitted, when base64 decoding fails on binary data when binary support is enacted, or when integration response mapping can't match any template and no default template is configured. If the response type is unspecified, this response defaults to the DEFAULT_5XX type.
AUTHORIZER_CONFIGURATION_ERROR	500	The gateway response for failing to connect to a custom or Amazon Cognito authorizer. If the response type is unspecified, this response defaults to the DEFAULT_5XX type.
AUTHORIZER_FAILURE	500	The gateway response when a custom or Amazon Cognito authorizer failed to authenticate the caller. If the response type is unspecified, this response defaults to the DEFAULT_5XX type.

Gateway response type	Default status code	Description
`BAD_REQUEST_PARAMETERS`	`400`	The gateway response when the request parameter cannot be validated according to an enabled request validator. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.
`BAD_REQUEST_BODY`	`400`	The gateway response when the request body cannot be validated according to an enabled request validator. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.

Gateway response type	Default status code	Description
`DEFAULT_4XX`	Null	The default gateway response for an unspecified response type with the status code of 4XX. Changing the status code of this fallback gateway response changes the status codes of all other 4XX responses to the new value. Resetting this status code to null reverts the status codes of all other 4XX responses to their original values. **Note** AWS WAF custom responses take precedence over custom gateway responses.
`DEFAULT_5XX`	Null	The default gateway response for an unspecified response type with a status code of 5XX. Changing the status code of this fallback gateway response changes the status codes of all other 5XX responses to the new value. Resetting this status code to null reverts the status codes of all other 5XX responses to their original values.

Gateway response type	Default status code	Description
`EXPIRED_TOKEN`	`403`	The gateway response for an AWS authentication token expired error. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.
`INTEGRATION_FAILURE`	`504`	The gateway response for an integration failed error. If the response type is unspecified, this response defaults to the `DEFAULT_5XX` type.
`INTEGRATION_TIMEOUT`	`504`	The gateway response for an integration timed out error. If the response type is unspecified, this response defaults to the `DEFAULT_5XX` type.
`INVALID_API_KEY`	`403`	The gateway response for an invalid API key submitted for a method requiring an API key. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.
`INVALID_SIGNATURE`	`403`	The gateway response for an invalid AWS signature error. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.

Gateway response type	Default status code	Description
MISSING_AUTHENTICATION_TOKEN	403	The gateway response for a missing authentication token error, including the cases when the client attempts to invoke an unsupported API method or resource. If the response type is unspecified, this response defaults to the DEFAULT_4XX type.
QUOTA_EXCEEDED	429	The gateway response for the usage plan quota exceeded error. If the response type is unspecified, this response defaults to the DEFAULT_4XX type.
REQUEST_TOO_LARGE	413	The gateway response for the request too large error. If the response type is unspecified, this response defaults to: HTTP content length exceeded 10485760 bytes.
RESOURCE_NOT_FOUND	404	The gateway response when API Gateway cannot find the specified resource after an API request passes authentication and authorization, except for API key authentication and authorization. If the response type is unspecified, this response defaults to the DEFAULT_4XX type.

Gateway response type	Default status code	Description
`THROTTLED`	429	The gateway response when usage plan-, method-, stage-, or account-level throttling limits exceeded. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.
`UNAUTHORIZED`	401	The gateway response when the custom or Amazon Cognito authorizer failed to authenticate the caller.
`UNSUPPORTED_MEDIA_TYPE`	415	The gateway response when a payload is of an unsupported media type, if strict passthrough behavior is enabled. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type.
`WAF_FILTERED`	403	The gateway response when a request is blocked by AWS WAF. If the response type is unspecified, this response defaults to the `DEFAULT_4XX` type. **Note** AWS WAF custom responses take precedence over custom gateway responses.

Enabling CORS for a REST API resource

Cross-origin resource sharing (CORS) is a browser security feature that restricts cross-origin HTTP requests that are initiated from scripts running in the browser.

Determining whether to enable CORS support

A *cross-origin* HTTP request is one that is made to:

- A different *domain* (for example, from `example.com` to `amazondomains.com`)
- A different *subdomain* (for example, from `example.com` to `petstore.example.com`)
- A different *port* (for example, from `example.com` to `example.com:10777`)
- A different *protocol* (for example, from `https://example.com` to `http://example.com`)

If you cannot access your API and receive an error message that contains `Cross-Origin Request Blocked`, you might need to enable CORS.

Cross-origin HTTP requests can be divided into two types: *simple* requests and *non-simple* requests.

Enabling CORS for a simple request

An HTTP request is *simple* if all of the following conditions are true:

- It is issued against an API resource that allows only `GET`, `HEAD`, and `POST` requests.
- If it is a `POST` method request, it must include an `Origin` header.
- The request payload content type is `text/plain`, `multipart/form-data`, or `application/x-www-form-urlencoded`.
- The request does not contain custom headers.
- Any additional requirements that are listed in the Mozilla CORS documentation for simple requests.

For simple cross-origin `POST` method requests, the response from your resource needs to include the header `Access-Control-Allow-Origin: '*'` or `Access-Control-Allow-Origin:`*`'origin'`*.

All other cross-origin HTTP requests are *non-simple* requests.

Enabling CORS for a non-simple request

If your API's resources receive non-simple requests, you must enable additional CORS support depending on your integration type.

Enabling CORS for non-proxy integrations

For these integrations, the CORS protocol requires the browser to send a preflight request to the server and wait for approval (or a request for credentials) from the server before sending the actual request. You must configure your API to send an appropriate response to the preflight request.

To create a preflight response:

1. Create an `OPTIONS` method with a mock integration.
2. Add the following response headers to the 200 method response:
   - `Access-Control-Allow-Headers`
   - `Access-Control-Allow-Methods`
   - `Access-Control-Allow-Origin`
3. Enter values for the response headers. To allow all origins, all methods, and common headers, use the following header values:
   - `Access-Control-Allow-Headers: 'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'`
   - `Access-Control-Allow-Methods: '*'`
   - `Access-Control-Allow-Origin: '*'`

After creating the preflight request, you must return the `Access-Control-Allow-Origin: '*'` or `Access-Control-Allow-Origin:`*`'origin'`* header for all CORS-enabled methods for at least all 200 responses.

Enabling CORS for non-proxy integrations using the AWS Management Console

You can use the AWS Management Console to enable CORS. API Gateway creates an `OPTIONS` method and adds the `Access-Control-Allow-Origin` header to your existing method integration responses. This doesn’t always work, and sometimes you need to manually modify the integration response to return the `Access-Control-Allow-Origin` header for all CORS-enabled methods for at least all 200 responses.

Enabling CORS support for proxy integrations

For a Lambda proxy integration or HTTP proxy integration, your backend is responsible for returning the `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, and `Access-Control-Allow-Headers` headers, because a proxy integration doesn't return an integration response.

The following example Lambda functions return the required CORS headers:

Node.js

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        headers: {
            "Access-Control-Allow-Headers" : "Content-Type",
            "Access-Control-Allow-Origin": "https://www.example.com",
            "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
        },
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

Python 3

```
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'headers': {
            'Access-Control-Allow-Headers': 'Content-Type',
            'Access-Control-Allow-Origin': 'https://www.example.com',
            'Access-Control-Allow-Methods': 'OPTIONS,POST,GET'
        },
        'body': json.dumps('Hello from Lambda!')
    }
```

Topics

- Enable CORS on a resource using the API Gateway console

- Enable CORS on a resource using the API Gateway import API
- Testing CORS

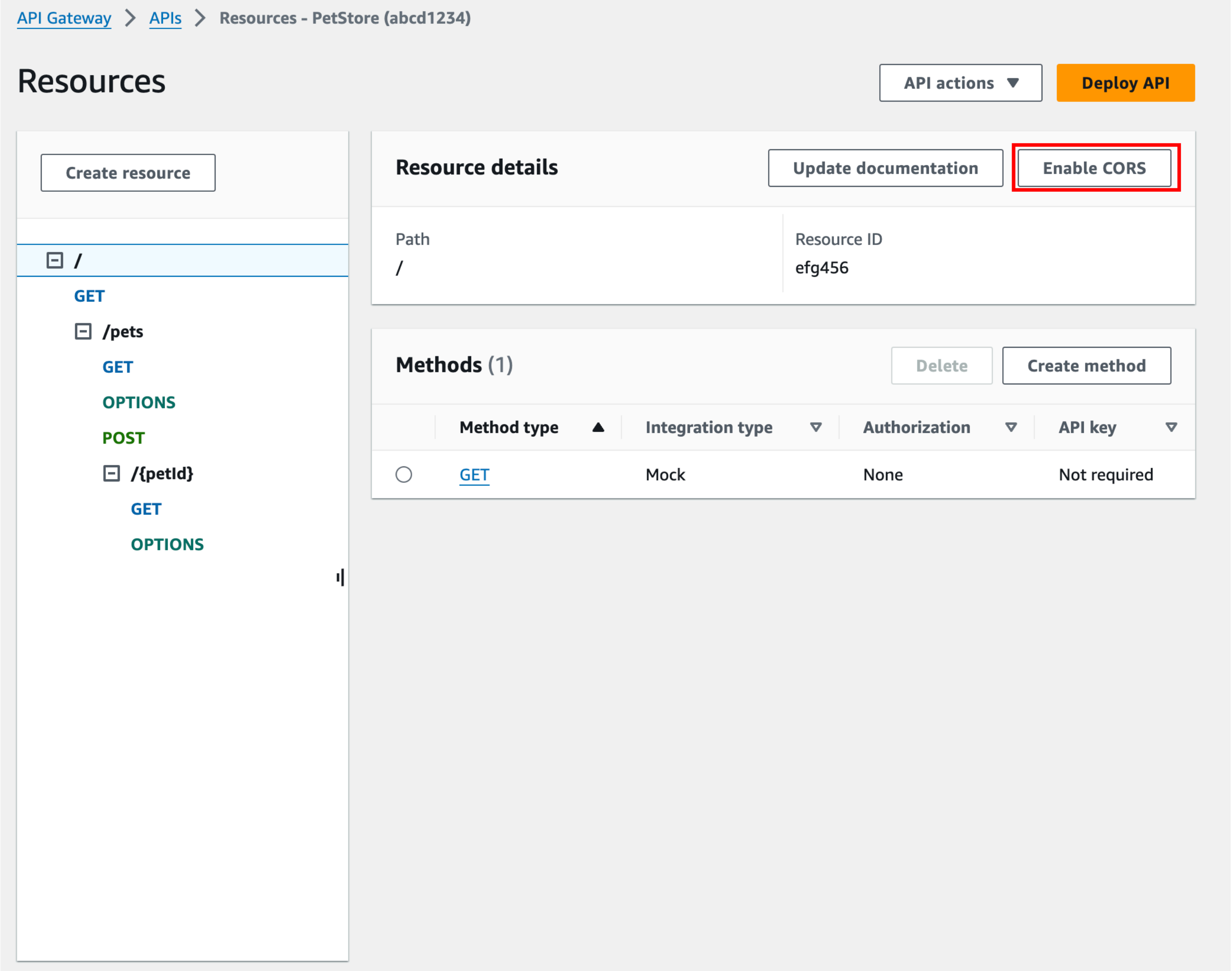
Enable CORS on a resource using the API Gateway console

You can use the API Gateway console to enable CORS support for one or all methods on a REST API resource that you have created.

> ⚠ **Important**
>
> Resources can contain child resources. Enabling CORS support for a resource and its methods does not recursively enable it for child resources and their methods.

To enable CORS support on a REST API resource

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an API.
3. Choose a resource under **Resources**.
4. In the **Resource details** section, choose **Enable CORS**.



Path	Resource ID
/	efg456

	Method type	Integration type	Authorization	API key
○	GET	Mock	None	Not required

5. In the **Enable CORS** box, do the following:

   a. (Optional) If you created a custom gateway response and want to enable CORS support for a response, select a gateway response.

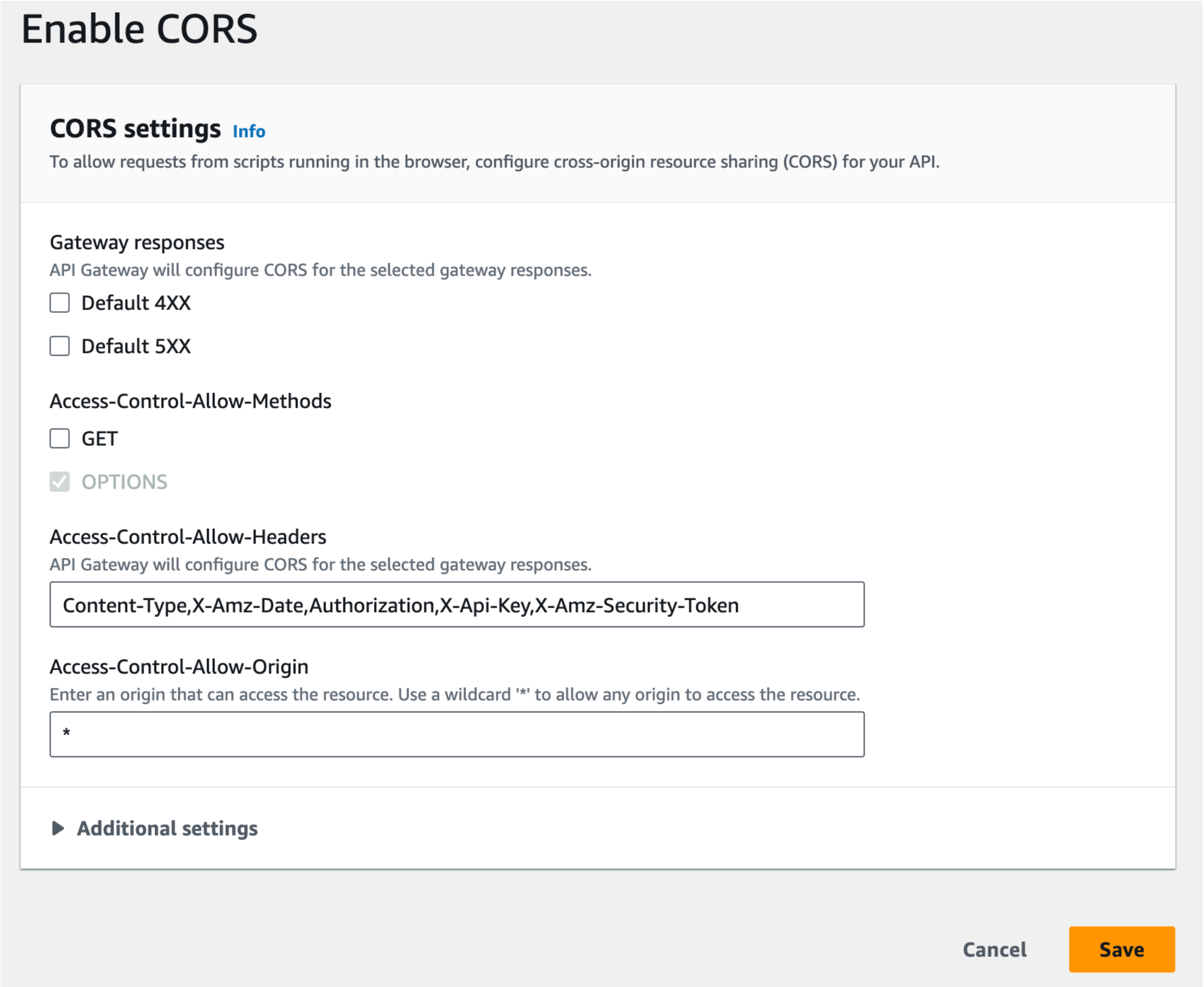
   b. Select each method to enable CORS support. The `OPTION` method must have CORS enabled.

      If you enable CORS support for an `ANY` method, CORS is enabled for all methods.

   c. In the **Access-Control-Allow-Headers** input field, enter a static string of a comma-separated list of headers that the client must submit in the actual request of the resource. Use the console-provided header list of `'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'` or specify your own headers.

d. Use the console-provided value of `'*'` as the **Access-Control-Allow-Origin** header value to allow access requests from all origins, or specify origins to be permitted to access the resource.

e. Choose **Save**.



Enable CORS

CORS settings Info

To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API.

Gateway responses

API Gateway will configure CORS for the selected gateway responses.

□ Default 4XX

□ Default 5XX

Access-Control-Allow-Methods

□ GET

☑ OPTIONS

Access-Control-Allow-Headers

API Gateway will configure CORS for the selected gateway responses.

Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token

Access-Control-Allow-Origin

Enter an origin that can access the resource. Use a wildcard '*' to allow any origin to access the resource.

*

Important

When applying the above instructions to the ANY method in a proxy integration, any applicable CORS headers will not be set. Instead, your backend must return the applicable CORS headers, such as `Access-Control-Allow-Origin`.

After CORS is enabled on the GET method, an OPTIONS method is added to the resource, if it is not already there. The 200 response of the OPTIONS method is automatically configured to return the three Access-Control-Allow-* headers to fulfill preflight handshakes. In addition, the actual (GET) method is also configured by default to return the Access-Control-Allow-Origin header in its 200 response as well. For other types of responses, you will need to manually configure them to return Access-Control-Allow-Origin' header with '*' or specific origins, if you do not want to return the Cross-origin access error.

After you enable CORS support on your resource, you must deploy or redeploy the API for the new settings to take effect. For more information, see the section called "Deploy a REST API (console)".

> **Note**
>
> If you cannot enable CORS support on your resource after following the procedure, we recommend that you compare your CORS configuration to the example API /pets resource. To learn how to create the example API, see the section called "Tutorial: Create a REST API by importing an example".

Enable CORS on a resource using the API Gateway import API

If you are using the API Gateway Import API, you can set up CORS support using an OpenAPI file. You must first define an OPTIONS method in your resource that returns the required headers.

> **Note**
>
> Web browsers expect Access-Control-Allow-Headers, and Access-Control-Allow-Origin headers to be set up in each API method that accepts CORS requests. In addition, some browsers first make an HTTP request to an OPTIONS method in the same resource, and then expect to receive the same headers.

Example Options method

The following example creates an OPTIONS method for a mock integration.

OpenAPI 3.0

```
/users:
  options:
    summary: CORS support
    description: |
      Enable CORS by returning correct headers
    tags:
    - CORS
    responses:
      200:
        description: Default response for CORS method
        headers:
          Access-Control-Allow-Origin:
            schema:
              type: "string"
          Access-Control-Allow-Methods:
            schema:
              type: "string"
          Access-Control-Allow-Headers:
            schema:
              type: "string"
        content: {}
    x-amazon-apigateway-integration:
      type: mock
      requestTemplates:
        application/json: "{\"statusCode\": 200}"
      responses:
        default:
          statusCode: "200"
          responseParameters:
            method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
            method.response.header.Access-Control-Allow-Methods: "'*'"
            method.response.header.Access-Control-Allow-Origin: "'*'"
```

OpenAPI 2.0

```
/users:
   options:
      summary: CORS support
      description: |
        Enable CORS by returning correct headers
      consumes:
        - "application/json"
      produces:
        - "application/json"
      tags:
        - CORS
      x-amazon-apigateway-integration:
        type: mock
        requestTemplates: "{\"statusCode\": 200}"
        responses:
          "default":
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Headers : "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Methods : "'*'"
              method.response.header.Access-Control-Allow-Origin : "'*'"
      responses:
        200:
          description: Default response for CORS method
          headers:
            Access-Control-Allow-Headers:
              type: "string"
            Access-Control-Allow-Methods:
              type: "string"
            Access-Control-Allow-Origin:
              type: "string"
```

Once you have configured the `OPTIONS` method for your resource, you can add the required headers to the other methods in the same resource that need to accept CORS requests.

1. Declare the **Access-Control-Allow-Origin** and **Headers** to the response types.

   OpenAPI 3.0

   ```
       responses:
         200:
           description: Default response for CORS method
           headers:
             Access-Control-Allow-Origin:
               schema:
                 type: "string"
             Access-Control-Allow-Methods:
               schema:
   ```

```
              type: "string"
            Access-Control-Allow-Headers:
              schema:
                type: "string"
          content: {}
```

OpenAPI 2.0

```
        responses:
          200:
            description: Default response for CORS method
            headers:
              Access-Control-Allow-Headers:
                type: "string"
              Access-Control-Allow-Methods:
                type: "string"
              Access-Control-Allow-Origin:
                type: "string"
```

2. In the `x-amazon-apigateway-integration` tag, set up the mapping for those headers to your static values:

   OpenAPI 3.0

```
      responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Headers: "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Methods: "'*'"
              method.response.header.Access-Control-Allow-Origin: "'*'"
            responseTemplates:
              application/json: |
                {}
```

   OpenAPI 2.0

```
          responses:
            "default":
              statusCode: "200"
              responseParameters:
```

```
              method.response.header.Access-Control-Allow-Headers : "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Methods : "'*'"
              method.response.header.Access-Control-Allow-Origin : "'*'"
```

Example API

The following example creates a complete API with an `OPTIONS` method and a `GET` method with an `HTTP` integration.

OpenAPI 3.0

```
openapi: "3.0.1"
info:
  title: "cors-api"
  description: "cors-api"
  version: "2024-01-16T18:36:01Z"
servers:
- url: "/{basePath}"
  variables:
    basePath:
      default: "/test"
paths:
  /:
    get:
      operationId: "GetPet"
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
              schema:
                type: "string"
          content: {}
      x-amazon-apigateway-integration:
        httpMethod: "GET"
        uri: "http://petstore.execute-api.us-east-1.amazonaws.com/petstore/pets"
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
        passthroughBehavior: "when_no_match"
        type: "http"
    options:
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
              schema:
                type: "string"
            Access-Control-Allow-Methods:
              schema:
                type: "string"
            Access-Control-Allow-Headers:
              schema:
                type: "string"
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Empty"
      x-amazon-apigateway-integration:
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Methods: "'GET,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Origin: "'*'"
        requestTemplates:
          application/json: "{\"statusCode\": 200}"
        passthroughBehavior: "when_no_match"
        type: "mock"
components:
  schemas:
    Empty:
      type: "object"
```

OpenAPI 2.0

```
swagger: "2.0"
info:
  description: "cors-api"
  version: "2024-01-16T18:36:01Z"
  title: "cors-api"
basePath: "/test"
schemes:
- "https"
paths:
  /:
    get:
      operationId: "GetPet"
      produces:
      - "application/json"
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
              type: "string"
      x-amazon-apigateway-integration:
        httpMethod: "GET"
        uri: "http://petstore.execute-api.us-east-1.amazonaws.com/petstore/pets"
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
        passthroughBehavior: "when_no_match"
        type: "http"
    options:
      consumes:
      - "application/json"
      produces:
      - "application/json"
      responses:
        "200":
          description: "200 response"
          schema:
            $ref: "#/definitions/Empty"
          headers:
            Access-Control-Allow-Origin:
              type: "string"
            Access-Control-Allow-Methods:
              type: "string"
            Access-Control-Allow-Headers:
              type: "string"
      x-amazon-apigateway-integration:
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Methods: "'GET,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Origin: "'*'"
        requestTemplates:
          application/json: "{\"statusCode\": 200}"
        passthroughBehavior: "when_no_match"
        type: "mock"
definitions:
  Empty:
    type: "object"
```

Testing CORS

You can test your API's CORS configuration by invoking your API, and checking the CORS headers in the response. The following `curl` command sends an OPTIONS request to a deployed API.

```
curl -v -X OPTIONS https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}
```

```
< HTTP/1.1 200 OK
< Date: Tue, 19 May 2020 00:55:22 GMT
< Content-Type: application/json
< Content-Length: 0
< Connection: keep-alive
< x-amzn-RequestId: a1b2c3d4-5678-90ab-cdef-abc123
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Headers: Content-Type,Authorization,X-Amz-Date,X-Api-Key,X-Amz-Security-Token
< x-amz-apigw-id: Abcd=
< Access-Control-Allow-Methods: DELETE,GET,HEAD,OPTIONS,PATCH,POST,PUT
```

The `Access-Control-Allow-Origin`, `Access-Control-Allow-Headers`, and `Access-Control-Allow-Methods` headers in the response show that the API supports CORS. For more information, see Enabling CORS for a REST API resource.

Working with binary media types for REST APIs

In API Gateway, the API request and response have a text or binary payload. A text payload is a UTF-8-encoded JSON string. A binary payload is anything other than a text payload. The binary payload can be, for example, a JPEG file, a GZip file, or an XML file. The API configuration required to support binary media depends on whether your API uses proxy or non-proxy integrations.

AWS Lambda proxy integrations

To handle binary payloads for AWS Lambda proxy integrations, you must base64-encode your function's response. You must also configure the binaryMediaTypes for your API. Your API's binaryMediaTypes configuration is a list of content types that your API treats as binary data. Example binary media types include image/png or application/octet-stream. You can use the wildcard character (*) to cover multiple media types. For example, */* includes all content types.

For example code, see the section called “Return binary media from a Lambda proxy integration”.

Non-proxy integrations

To handle binary payloads for non-proxy integrations, you add the media types to the binaryMediaTypes list of the RestApi resource. Your API's binaryMediaTypes configuration is a list of content types that your API treats as binary data. Alternatively, you can set the contentHandling properties on the Integration and the IntegrationResponse resources. The contentHandling value can be CONVERT_TO_BINARY, CONVERT_TO_TEXT, or undefined.

Depending on the contentHandling value, and whether the Content-Type header of the response or the Accept header of the incoming request matches an entry in the binaryMediaTypes list, API Gateway can encode the raw binary bytes as a base64-encoded string, decode a base64-encoded string back to its raw bytes, or pass the body through without modification.

You must configure the API as follows to support binary payloads for your API in API Gateway:

- Add the desired binary media types to the binaryMediaTypes list on the RestApi resource. If this property and the contentHandling property are not defined, the payloads are handled as UTF-8 encoded JSON strings.
- Address the contentHandling property of the Integration resource.

    - To have the request payload converted from a base64-encoded string to its binary blob, set the property to CONVERT_TO_BINARY.
    - To have the request payload converted from a binary blob to a base64-encoded string, set the property to CONVERT_TO_TEXT.
    - To pass the payload through without modification, leave the property undefined. To pass a binary payload through without modification, you must also ensure that the Content-Type matches one of the binaryMediaTypes entries, and that passthrough behaviors are enabled for the API.
- Set the contentHandling property of the IntegrationResponse resource. The contentHandling property, Accept header in client requests, and your API's binaryMediaTypes combined determine how API Gateway handles content type conversions. For details, see the section called "Content type conversions in API Gateway".

> **Important**
>
> When a request contains multiple media types in its Accept header, API Gateway honors only the first Accept media type. If you can't control the order of the Accept media types and the media type of your binary content isn't the first in the list, add the first Accept media type in the binaryMediaTypes list of your API. API Gateway handles all content types in this list as binary.
>
> For example, to send a JPEG file using an <img> element in a browser, the browser might send Accept:image/webp,image/*,*/*;q=0.8 in a request. By adding image/webp to the binaryMediaTypes list, the endpoint receives the JPEG file as binary.

For detailed information about how API Gateway handles the text and binary payloads, see Content type conversions in API Gateway.

Content type conversions in API Gateway

The combination of your API's binaryMediaTypes, the headers in client requests, and the integration contentHandling property determine how API Gateway encodes payloads.

The following table shows how API Gateway converts the request payload for specific configurations of a request's Content-Type header, the binaryMediaTypes list of a RestApi resource, and the contentHandling property value of the Integration resource.

API request content type conversions in API Gateway

Method request payload	Request `Content-Type` header	`binaryMediaTypes`	`contentHandling`	Integration request payload
Text data	Any data type	Undefined	Undefined	UTF8-encoded string
Text data	Any data type	Undefined	`CONVERT_TO_BINARY`	Base64-decoded binary blob
Text data	Any data type	Undefined	`CONVERT_TO_TEXT`	UTF8-encoded string
Text data	A text data type	Set with matching media types	Undefined	Text data
Text data	A text data type	Set with matching media types	`CONVERT_TO_BINARY`	Base64-decoded binary blob
Text data	A text data type	Set with matching media types	`CONVERT_TO_TEXT`	Text data
Binary data	A binary data type	Set with matching media types	Undefined	Binary data
Binary data	A binary data type	Set with matching media types	`CONVERT_TO_BINARY`	Binary data
Binary data	A binary data type	Set with matching media types	`CONVERT_TO_TEXT`	Base64-encoded string

The following table shows how API Gateway converts the response payload for specific configurations of a request's Accept header, the binaryMediaTypes list of a RestApi resource, and the contentHandling property value of the IntegrationResponse resource.

Important

When a request contains multiple media types in its Accept header, API Gateway honors only the first Accept media type. If you can't control the order of the Accept media types and the media type of your binary content isn't the first in the list, add the first Accept media type in the binaryMediaTypes list of your API. API Gateway handles all content types in this list as binary.

For example, to send a JPEG file using an <img> element in a browser, the browser might send Accept:image/webp,image/*,*/*;q=0.8 in a request. By adding image/webp to the binaryMediaTypes list, the endpoint receives the JPEG file as binary.

API Gateway response content type conversions

Integration response payload	Request Accept header	binaryMediaTypes	contentHandling	Method response payload
Text or binary data	A text type	Undefined	Undefined	UTF8-encoded string
Text or binary data	A text type	Undefined	CONVERT_TO_BINARY	Base64-decoded blob
Text or binary data	A text type	Undefined	CONVERT_TO_TEXT	UTF8-encoded string
Text data	A text type	Set with matching media types	Undefined	Text data
Text data	A text type	Set with matching media types	CONVERT_TO_BINARY	Base64-decoded blob

Integration response payload	Request Accept header	binaryMediaTypes	contentHandling	Method response payload
Text data	A text type	Set with matching media types	CONVERT_TO_TEXT	UTF8-encoded string
Text data	A binary type	Set with matching media types	Undefined	Base64-decoded blob
Text data	A binary type	Set with matching media types	CONVERT_TO_BINARY	Base64-decoded blob
Text data	A binary type	Set with matching media types	CONVERT_TO_TEXT	UTF8-encoded string
Binary data	A text type	Set with matching media types	Undefined	Base64-encoded string
Binary data	A text type	Set with matching media types	CONVERT_TO_BINARY	Binary data
Binary data	A text type	Set with matching media types	CONVERT_TO_TEXT	Base64-encoded string
Binary data	A binary type	Set with matching media types	Undefined	Binary data
Binary data	A binary type	Set with matching media types	CONVERT_TO_BINARY	Binary data

Integration response payload	Request Accept header	binaryMediaTypes	contentHandling	Method response payload
Binary data	A binary type	Set with matching media types	CONVERT_TO_TEXT	Base64-encoded string

When converting a text payload to a binary blob, API Gateway assumes that the text data is a base64-encoded string and outputs the binary data as a base64-decoded blob. If the conversion fails, it returns a 500 response, which indicates an API configuration error. You don't provide a mapping template for such a conversion, although you must enable the passthrough behaviors on the API.

When converting a binary payload to a text string, API Gateway always applies a base64 encoding on the binary data. You can define a mapping template for such a payload, but can only access the base64-encoded string in the mapping template through `$input.body`, as shown in the following excerpt of an example mapping template.

```
{
    "data": "$input.body"
}
```

To have the binary payload passed through without modification, you must enable the passthrough behaviors on the API.

Enabling binary support using the API Gateway console

The section explains how to enable binary support using the API Gateway console. As an example, we use an API that is integrated with Amazon S3. We focus on the tasks to set the supported media types and to specify how the payload should be handled. For detailed information on how to create an API integrated with Amazon S3, see Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway.

To enable binary support by using the API Gateway console

1. Set binary media types for the API:

   a. Create a new API or choose an existing API. For this example, we name the API `FileMan`.

b. Under the selected API in the primary navigation panel, choose **API settings**.

c. In the **API settings** pane, choose **Manage media types** in the **Binary Media Types** section.

d. Choose **Add binary media type**.

e. Enter a required media type, for example, **`image/png`**, in the input text field. If needed, repeat this step to add more media types. To support all binary media types, specify `*/*`.

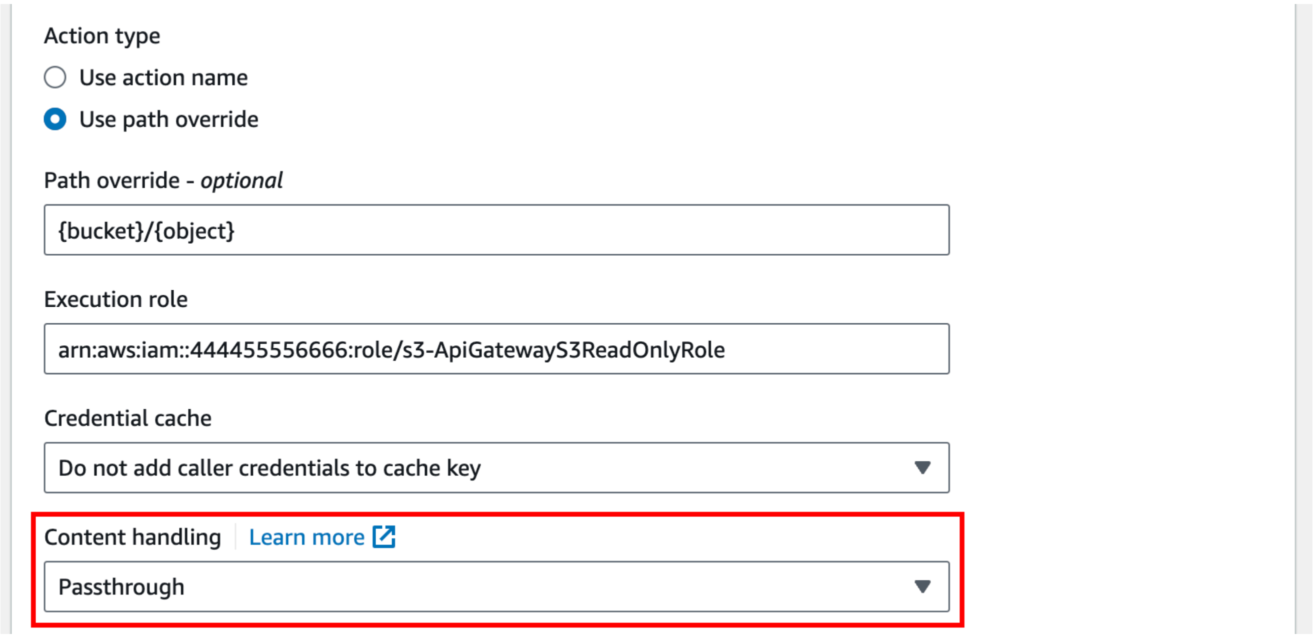
f. Choose **Save changes**.

2. Set how message payloads are handled for the API method:

a. Create a new or choose an existing resource in the API. For this example, we use the `/{folder}/{item}` resource.

b. Create a new or choose an existing method on the resource. As an example, we use the `GET /{folder}/{item}` method integrated with the `Object GET` action in Amazon S3.

c. For **Content handling**, choose an option.



Action type

○ Use action name

● Use path override

Path override - *optional*

{bucket}/{object}

Execution role

arn:aws:iam::444455556666:role/s3-ApiGatewayS3ReadOnlyRole

Credential cache

Do not add caller credentials to cache key ▼

Content handling | Learn more

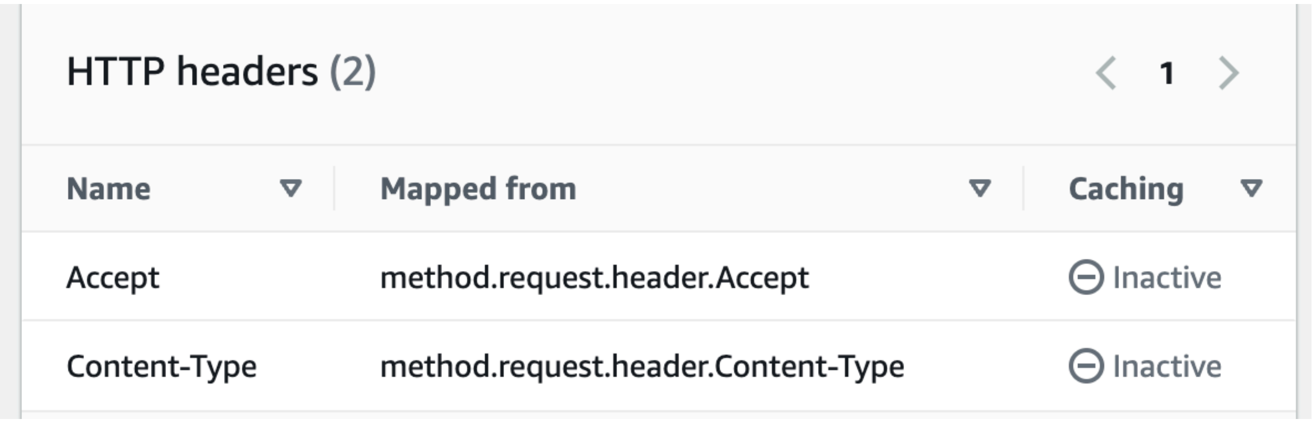
Passthrough ▼

Choose **Passthrough** if you don't want to convert the body when the client and backend accepts the same binary format. Choose **Convert to text** to convert the binary body to a base64-encoded string when, for example, the backend requires that a binary request payload is passed in as a JSON property. And choose **Convert to binary** when the client submits a base64-encoded string and the backend requires the original binary format, or when the endpoint returns a base64-encoded string and the client accepts only the binary output.

d. For **Request body passthrough**, choose **When there are no templates defined (recommended)** to enable the passthrough behavior on the request body.

   You could also choose **Never**. This means that the API will reject data with content-types that do not have a mapping template.

e. Preserve the incoming request's `Accept` header in the integration request. You should do this if you've set `contentHandling` to `passthrough` and want to override that setting at runtime.



HTTP headers (2)

Name	Mapped from	Caching
Accept	method.request.header.Accept	⊖ Inactive
Content-Type	method.request.header.Content-Type	⊖ Inactive

f. For conversion to text, define a mapping template to put the base64-encoded binary data into the required format.

   An example of a mapping template to convert to text is the following:

```
{
  "operation": "thumbnail",
  "base64Image": "$input.body"
}
```

   The format of this mapping template depends on the endpoint requirements of the input.

g. Choose **Save**.

Enabling binary support using the API Gateway REST API

The following tasks show how to enable binary support using the API Gateway REST API calls.

Topics

- Add and update supported binary media types to an API

- Configure request payload conversions
- Configure response payload conversions
- Convert binary data to text data
- Convert text data to a binary payload
- Pass through a binary payload

Add and update supported binary media types to an API

To enable API Gateway to support a new binary media type, you must add the binary media type to the `binaryMediaTypes` list of the `RestApi` resource. For example, to have API Gateway handle JPEG images, submit a `PATCH` request to the `RestApi` resource:

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/image~1jpeg"
  }
 ]
}
```

The MIME type specification of `image/jpeg` that is part of the `path` property value is escaped as `image~1jpeg`.

To update the supported binary media types, replace or remove the media type from the `binaryMediaTypes` list of the `RestApi` resource. For example, to change binary support from JPEG files to raw bytes, submit a `PATCH` request to the `RestApi` resource, as follows:

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [{
    "op" : "replace",
    "path" : "/binaryMediaTypes/image~1jpeg",
    "value" : "application/octet-stream"
  },
  {
    "op" : "remove",
    "path" : "/binaryMediaTypes/image~1jpeg"
  }]
}
```

Configure request payload conversions

If the endpoint requires a binary input, set the `contentHandling` property of the `Integration` resource to `CONVERT_TO_BINARY`. To do so, submit a `PATCH` request, as follows:

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/integration

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/contentHandling",
    "value" : "CONVERT_TO_BINARY"
  }]
}
```

Configure response payload conversions

If the client accepts the result as a binary blob instead of a base64-encoded payload returned from the endpoint, set the `contentHandling` property of the `IntegrationResponse` resource to `CONVERT_TO_BINARY`. To do this, submit a `PATCH` request, as follows:

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/integration/
responses/<status_code>

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/contentHandling",
    "value" : "CONVERT_TO_BINARY"
  }]
}
```

Convert binary data to text data

To send binary data as a JSON property of the input to AWS Lambda or Kinesis through API Gateway, do the following:

1. Enable the binary payload support of the API by adding the new binary media type of `application/octet-stream` to the API's `binaryMediaTypes` list.

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/application~1octet-stream"
  }
 ]
}
```

2. Set `CONVERT_TO_TEXT` on the `contentHandling` property of the `Integration` resource and provide a mapping template to assign the base64-encoded string of the binary data to a JSON property. In the following example, the JSON property is body and `$input.body` holds the base64-encoded string.

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration

{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_TEXT"
    },
    {
      "op" : "add",
      "path" : "/requestTemplates/application~1octet-stream",
      "value" : "{\"body\": \"$input.body\"}"
    }
  ]
}
```

Convert text data to a binary payload

Suppose a Lambda function returns an image file as a base64-encoded string. To pass this binary output to the client through API Gateway, do the following:

1. Update the API's `binaryMediaTypes` list by adding the binary media type of `application/octet-stream`, if it is not already in the list.

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/application~1octet-stream",
  }]
}
```

2. Set the `contentHandling` property on the `Integration` resource to `CONVERT_TO_BINARY`. Do not define a mapping template. If you don't define a mapping template, API Gateway invokes the passthrough template to return the base64-decoded binary blob as the image file to the client.

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration/responses/<status_code>

{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_BINARY"
    }
  ]
}
```

Pass through a binary payload

To store an image in an Amazon S3 bucket using API Gateway, do the following:

1. Update the API's `binaryMediaTypes` list by adding the binary media type of `application/octet-stream`, if it isn't already in the list.

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/application~1octet-stream"
    }
  ]
}
```

2. On the `contentHandling` property of the `Integration` resource, set `CONVERT_TO_BINARY`. Set `WHEN_NO_MATCH` as the `passthroughBehavior` property value without defining a mapping template. This enables API Gateway to invoke the passthrough template.

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration

{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_BINARY"
    },
    {
      "op" : "replace",
      "path" : "/passthroughBehaviors",
      "value" : "WHEN_NO_MATCH"
    }
  ]
}
```

Import and export content encodings

To import the `binaryMediaTypes` list on a RestApi, use the following API Gateway extension to the API's OpenAPI definition file. The extension is also used to export the API settings.

- x-amazon-apigateway-binary-media-types property

To import and export the `contentHandling` property value on an `Integration` or `IntegrationResponse` resource, use the following API Gateway extensions to the OpenAPI definitions:

- x-amazon-apigateway-integration object
- x-amazon-apigateway-integration.response object

Examples of binary support

The following examples demonstrate how to access a binary file in Amazon S3 or AWS Lambda through an API Gateway API.

Topics

- Return binary media from a Lambda proxy integration
- Access binary files in Amazon S3 through an API Gateway API
- Access binary files in Lambda using an API Gateway API

Return binary media from a Lambda proxy integration

To return binary media from an AWS Lambda proxy integration, base64 encode the response from your Lambda function. You must also configure your API's binary media types. The payload size limit is 10 MB.

> **Note**
>
> To use a web browser to invoke an API with this example integration, set your API's binary media types to `*/*`. API Gateway uses the first `Accept` header from clients to determine if a response should return binary media. To return binary media when you can't control the order of `Accept` header values, such as requests from a browser, set your API's binary media types to `*/*` (for all content types).

The following example Lambda function can return a binary image from Amazon S3 or text to clients. The function's response includes a `Content-Type` header to indicate to the client the type of data that it returns. The function conditionally sets the `isBase64Encoded` property in its response, depending on the type of data that it returns.

Node.js

```
import { S3Client, GetObjectCommand } from "@aws-sdk/client-s3"

const client = new S3Client({region: 'us-east-2'});

export const handler = async (event) => {
  var randomint = function(max) {
    return Math.floor(Math.random() * max);
  }
  var number = randomint(2);
  if (number == 1){
    const input = {
      "Bucket" : "bucket-name",
      "Key" : "image.png"
      }
    try {
      const command = new GetObjectCommand(input)
      const response = await client.send(command);
      var str = await response.Body.transformToByteArray();
    } catch (err) {
      console.error(err);
    }
    const base64body = Buffer.from(str).toString('base64');
    return {
      'headers': { "Content-Type": "image/png" },
      'statusCode': 200,
      'body': base64body,
      'isBase64Encoded': true
      }
    } else {
        return {
        'headers': { "Content-Type": "text/html" },
        'statusCode': 200,
        'body': "<h1>This is text</h1>",
        }
    }
}
```

Python

```
import base64
import boto3
import json
import random

s3 = boto3.client('s3')

def lambda_handler(event, context):
    number = random.randint(0,1)
    if number == 1:
        response = s3.get_object(
            Bucket='bucket-name',
            Key='image.png',
        )
        image = response['Body'].read()
        return {
            'headers': { "Content-Type": "image/png" },
            'statusCode': 200,
            'body': base64.b64encode(image).decode('utf-8'),
            'isBase64Encoded': True
        }
    else:
        return {
            'headers': { "Content-type": "text/html" },
            'statusCode': 200,
            'body': "<h1>This is text</h1>",
        }
```

To learn more about binary media types, see Working with binary media types for REST APIs.

Access binary files in Amazon S3 through an API Gateway API

The following examples show the OpenAPI file used to access images in Amazon S3, how to download an image from Amazon S3, and how to upload an image to Amazon S3.

Topics

OpenAPI file of a sample API to access images in Amazon S3

The following OpenAPI file shows a sample API that illustrates downloading an image file from Amazon S3 and uploading an image file to Amazon S3. This API exposes the `GET /s3?key={file-name}` and `PUT /s3?key={file-name}` methods for downloading and uploading a specified image file. The GET method returns the image file as a base64-encoded string as part of a JSON output, following the supplied mapping template, in a 200 OK response. The PUT method takes a raw binary blob as input and returns a 200 OK response with an empty payload.

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-10-21T17:26:28Z",
      "title": "ApiName"
   },
   "paths": {
      "/s3": {
         "get": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               },
               "requestParameters": {
                  "integration.request.path.key": "method.request.querystring.key"
               },
               "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "GET",
               "type": "aws"
            }
         },
         "put": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     },
                     "application/octet-stream": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               },
               "requestParameters": {
                  "integration.request.path.key": "method.request.querystring.key"
               },
               "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "PUT",
               "type": "aws",
               "contentHandling": "CONVERT_TO_BINARY"
            }
         }
      }
   },
   "x-amazon-apigateway-binary-media-types": [
      "application/octet-stream",
      "image/jpeg"
   ],
   "servers": [
      {
         "url": "https://abcdefghi.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/v1"
            }
         }
      }
   ],
   "components": {
      "schemas": {
         "Empty": {
            "type": "object",
            "title": "Empty Schema"
         }
      }
   }
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-21T17:26:28Z",
    "title": "ApiName"
  },
  "host": "abcdefghi.execute-api.us-east-1.amazonaws.com",
  "basePath": "/v1",
  "schemes": [
    "https"
  ],
  "paths": {
    "/s3": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"            }
          },
          "requestParameters": {
            "integration.request.path.key": "method.request.querystring.key"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json", "application/octet-stream"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.path.key": "method.request.querystring.key"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "PUT",
          "type": "aws",
          "contentHandling" : "CONVERT_TO_BINARY"
        }
      }
    }
  },
  "x-amazon-apigateway-binary-media-types" : ["application/octet-stream", "image/
jpeg"],
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

Download an image from Amazon S3

To download an image file (`image.jpg`) as a binary blob from Amazon S3:

```
GET /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/octet-stream
```

The successful response looks like this:

```
200 OK HTTP/1.1

[raw bytes]
```

The raw bytes are returned because the `Accept` header is set to a binary media type of `application/octet-stream` and binary support is enabled for the API.

Alternatively, to download an image file (`image.jpg`) as a base64-encoded string (formatted as a JSON property) from Amazon S3, add a response template to the 200 integration response, as shown in the following bold-faced OpenAPI definition block:

```
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "{\n   \"image\": \"$input.body\"\n}"
              }
            }
          },
```

The request to download the image file looks like the following:

```
GET /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

The successful response looks like the following:

```
200 OK HTTP/1.1

{
  "image": "W3JhdyBieXRlc10="
}
```

Upload an image to Amazon S3

To upload an image file (`image.jpg`) as a binary blob to Amazon S3:

```
PUT /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/octet-stream
Accept: application/json
```

```
[raw bytes]
```

The successful response looks like the following:

```
200 OK HTTP/1.1
```

To upload an image file (`image.jpg`) as a base64-encoded string to Amazon S3:

```
PUT /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json

W3JhdyBieXRlc10=
```

The input payload must be a base64-encoded string because the `Content-Type` header value is set to `application/json`. The successful response looks like the following:

```
200 OK HTTP/1.1
```

Access binary files in Lambda using an API Gateway API

The following example demonstrates how to access a binary file in AWS Lambda through an API Gateway API. The sample API is presented in an OpenAPI file. The code example uses the API Gateway REST API calls.

Topics

OpenAPI file of a sample API to access images in Lambda

The following OpenAPI file shows an example API that illustrates downloading an image file from Lambda and uploading an image file to Lambda.

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-10-21T17:26:28Z",
      "title": "ApiName"
   },
   "paths": {
      "/lambda": {
         "get": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
               "type": "AWS",
               "credentials": "arn:aws:iam::123456789012:role/Lambda",
               "httpMethod": "POST",
               "requestTemplates": {
                  "application/json": "{\n   \"imageKey\":
 \"$input.params('key')\"\n}"
               },
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200",
                     "responseTemplates": {
                        "application/json": "{\n   \"image\": \"$input.body\"\n}"
                     }
                  }
               }
            }
         },
         "put": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     },
                     "application/octet-stream": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
               "type": "AWS",
               "credentials": "arn:aws:iam::123456789012:role/Lambda",
               "httpMethod": "POST",
               "contentHandling": "CONVERT_TO_TEXT",
               "requestTemplates": {
                  "application/json": "{\n   \"imageKey\": \"$input.params('key')\",
 \"image\": \"$input.body\"\n}"
               },
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               }
            }
         }
      }
   },
   "x-amazon-apigateway-binary-media-types": [
      "application/octet-stream",
      "image/jpeg"
   ],
   "servers": [
      {
         "url": "https://abcdefghi.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/v1"
            }
         }
      }
   ],
   "components": {
      "schemas": {
         "Empty": {
            "type": "object",
            "title": "Empty Schema"
         }
      }
   }
```

```
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-21T17:26:28Z",
    "title": "ApiName"
  },
  "host": "abcdefghi.execute-api.us-east-1.amazonaws.com",
  "basePath": "/v1",
  "schemes": [
    "https"
  ],
  "paths": {
    "/lambda": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
          "type": "AWS",
          "credentials": "arn:aws:iam::123456789012:role/Lambda",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n   \"imageKey\": \"$input.params('key')\"\n}"
          },
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "{\n   \"image\": \"$input.body\"\n}"
              }
            }
          }
        }
      },
      "put": {
        "produces": [
          "application/json", "application/octet-stream"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
          "type": "AWS",
          "credentials": "arn:aws:iam::123456789012:role/Lambda",
          "httpMethod": "POST",
          "contentHandling" : "CONVERT_TO_TEXT",
          "requestTemplates": {
            "application/json": "{\n   \"imageKey\": \"$input.params('key')\",
 \"image\": \"$input.body\"\n}"
          },
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"
            }
          }
        }
      }
    }
  },
  "x-amazon-apigateway-binary-media-types" : ["application/octet-stream", "image/
jpeg"],
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

Download an image from Lambda

To download an image file (`image.jpg`) as a binary blob from Lambda:

```
GET /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/octet-stream
```

The successful response looks like the following:

```
200 OK HTTP/1.1

[raw bytes]
```

To download an image file (`image.jpg`) as a base64-encoded string (formatted as a JSON property) from Lambda:

```
GET /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

The successful response looks like the following:

```
200 OK HTTP/1.1

{
  "image": "W3JhdyBieXRlc10="
}
```

Upload an image to Lambda

To upload an image file (`image.jpg`) as a binary blob to Lambda:

```
PUT /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/octet-stream
Accept: application/json

[raw bytes]
```

The successful response looks like the following:

```
200 OK
```

To upload an image file (`image.jpg`) as a base64-encoded string to Lambda:

```
PUT /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
```

```
Accept: application/json

W3JhdyBieXRlc10=
```

The successful response looks like the following:

```
200 OK
```

Invoking a REST API in Amazon API Gateway

To call a deployed API, clients submit requests to the URL for the API Gateway component service for API execution, known as `execute-api`.

The base URL for REST APIs is in the following format:

```
https://restapi_id.execute-api.region.amazonaws.com/stage_name/
```

where *`restapi_id`* is the API identifier, *`region`* is the AWS Region, and *`stage_name`* is the stage name of the API deployment.

⚠ **Important**

Before you can invoke an API, you must deploy it in API Gateway. For instructions on deploying an API, see Deploying a REST API in Amazon API Gateway.

Topics

- Obtaining an API's invoke URL
- Invoking an API
- Use the API Gateway console to test a REST API method
- Call REST API through generated SDKs
- How to invoke a private API

Obtaining an API's invoke URL

You can use the console, the AWS CLI, or an exported OpenAPI definition to obtain an API's invoke URL.

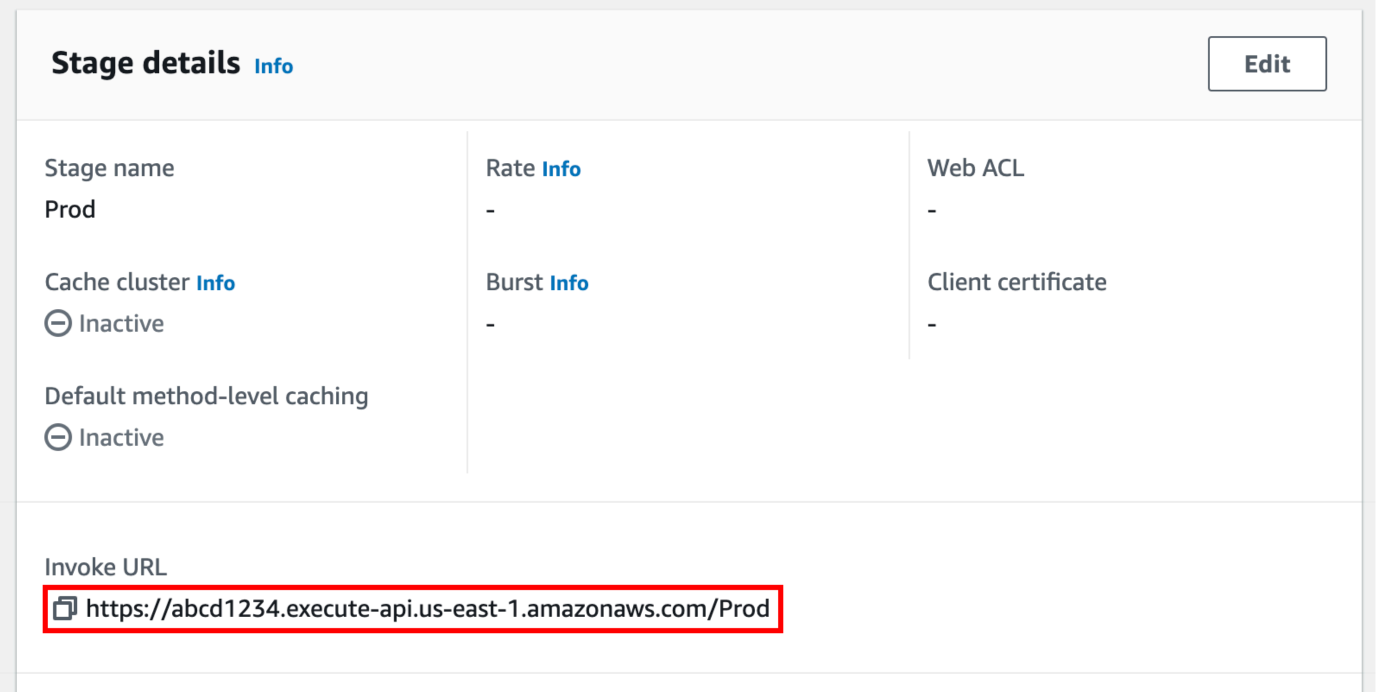
Obtaining an API's invoke URL using the console

The following procedure shows how to obtain an API's invoke URL in the REST API console.

To obtain an API's invoke URL using the REST API console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a deployed API.
3. From the main navigation pane, choose **Stage**.
4. Under **Stage details**, choose the copy icon to copy your API's invoke URL.

   This URL is for the root resource of your API.



Stage details Info

Edit

Stage name
Prod

Rate Info
-

Web ACL
-

Cache cluster Info
⊖ Inactive

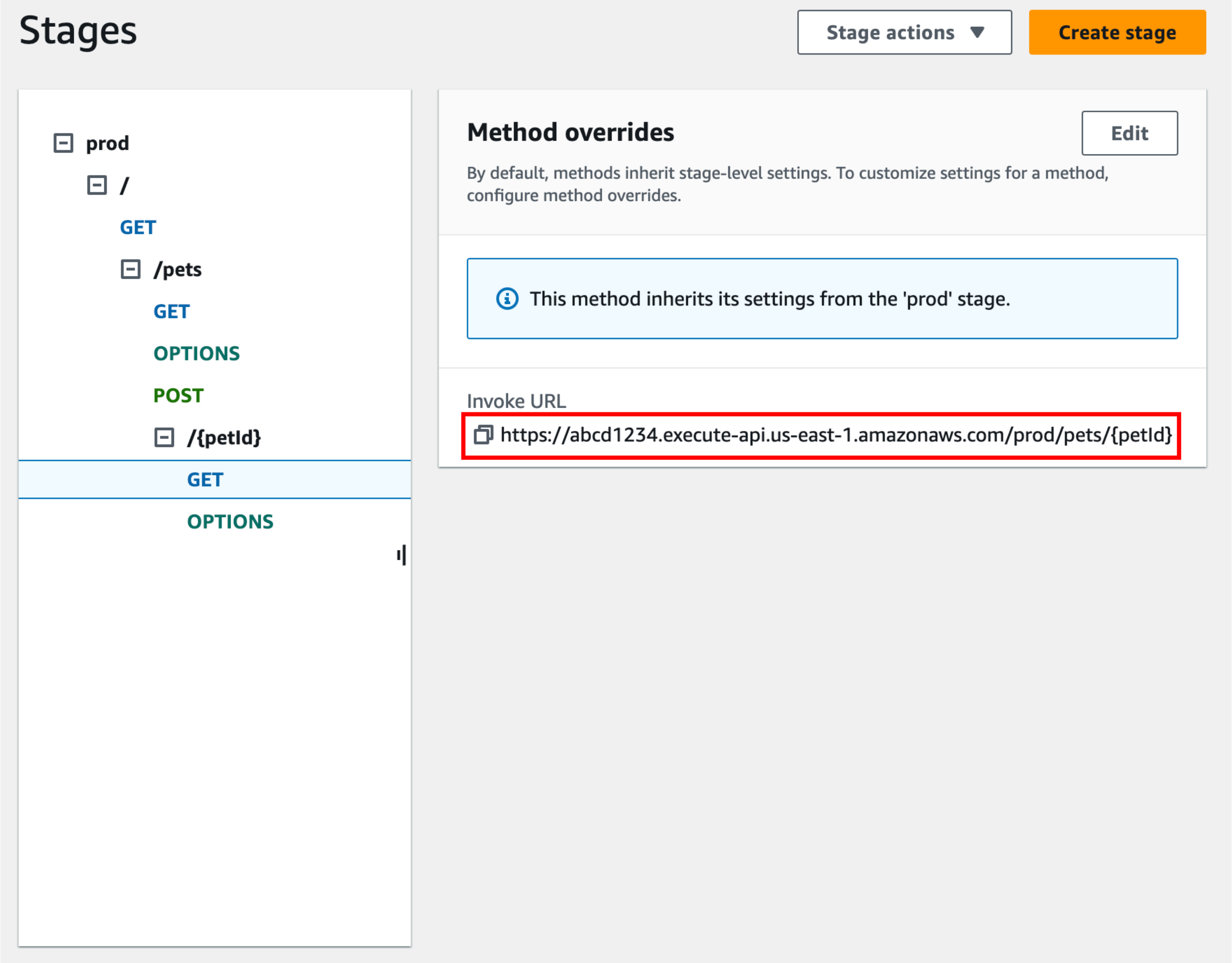
Burst Info
-

Client certificate
-

Default method-level caching
⊖ Inactive

Invoke URL
https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod

5. To obtain an API's invoke URL for another resource in your API, expand the stage under the secondary navigation pane, and then choose a method.
6. Choose the copy icon to copy your API's resource-level invoke URL.



By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.

This method inherits its settings from the 'prod' stage.

Invoke URL

https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/pets/{petId}

Obtaining an API's invoke URL using the AWS CLI

The following procedure shows how to obtain an API's invoke URL using the AWS CLI.

To obtain an API's invoke URL using the AWS CLI

1. Use the following command to obtain the `rest-api-id`. This command returns all `rest-api-id` values in your Region. For more information, see get-rest-apis.

   ```
   aws apigateway get-rest-apis
   ```

2. Replace the example `rest-api-id` with your `rest-api-id`, replace the example *`{stage-name}`* with your *`{stage-name}`*, and replace the *`{region}`*, with your Region.

```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

Obtaining an API's invoke URL using the exported OpenAPI definition file of the API

You can also construct the root URL by combining the `host` and `basePath` fields of an exported OpenAPI definition file of the API. For instructions on how to export your API, see the section called "Export a REST API".

Invoking an API

You can call your deployed API using the browser, curl, or other applications, like Postman.

Additionally, you can use the API Gateway console to test an API call. Test uses the API Gateway's `TestInvoke` feature, which allows API testing before the API is deployed. For more information, see the section called "Use the console to test a REST API method".

> **Note**
>
> Query string parameter values in an invocation URL cannot contain `%%`.

Invoking an API using a web browser

If your API permits anonymous access, you can use any web browser to invoke any GET method. Enter the complete invocation URL in the browser's address bar.

For other methods or any authentication-required calls, you must specify a payload or sign the requests. You can handle these in a script behind an HTML page or in a client application using one of the AWS SDKs.

Invoking an API using curl

You can use a tool like curl in your terminal to call your API. The following example curl command invokes the GET method on the `getUsers` resource of the `prod` stage of an API.

Linux or Macintosh

```
curl -X GET 'https://b123abcde4.execute-api.us-west-2.amazonaws.com/prod/getUsers'
```

Windows

```
curl -X GET "https://b123abcde4.execute-api.us-west-2.amazonaws.com/prod/getUsers"
```

Use the API Gateway console to test a REST API method

Use the API Gateway console to test a REST API method.

Topics

- Prerequisites
- Test a method with the API Gateway console

Prerequisites

- You must specify the settings for the methods you want to test. Follow the instructions in Set up REST API methods in API Gateway.

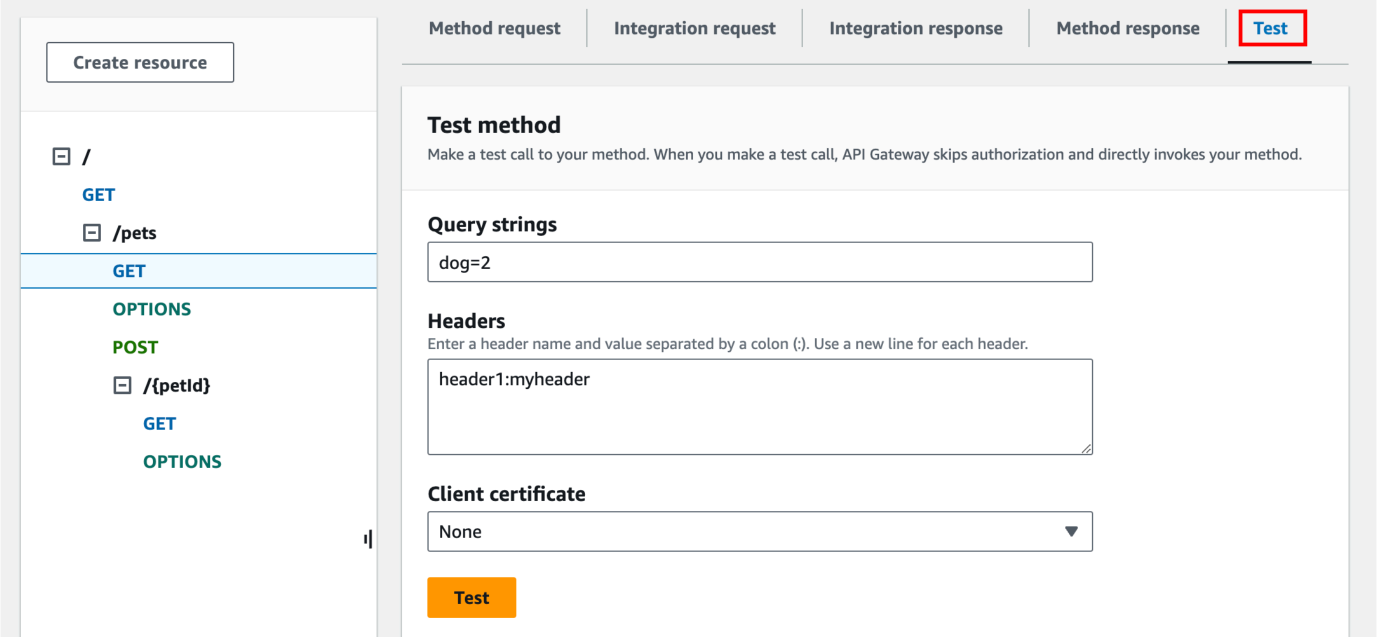
Test a method with the API Gateway console

⚠ **Important**

Testing methods with the API Gateway console might result in changes to resources that cannot be undone. Testing a method with the API Gateway console is the same as calling the method outside of the API Gateway console. For example, if you use the API Gateway console to call a method that deletes an API's resources, if the method call is successful, the API's resources will be deleted.

To test a method

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the **Resources** pane, choose the method you want to test.
4. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

dog=2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:myheader

Client certificate

None

Enter values in any of the displayed boxes (such as **Query strings**, **Headers**, and **Request body**). The console includes these values in the method request in default application/json form.

For additional options you might need to specify, contact the API owner.

5. Choose **Test**. The following information will be displayed:

- **Request** is the resource's path that was called for the method.
- **Status** is the response's HTTP status code.
- **Latency** is the time between the receipt of the request from the caller and the returned response.
- **Response body** is the HTTP response body.
- **Response headers** are the HTTP response headers.

> **Tip**
>
> Depending on the mapping, the HTTP status code, response body, and response headers might be different from those sent from the Lambda function, HTTP proxy, or AWS service proxy.

- **Log** shows the simulated Amazon CloudWatch Logs entries that would have been written if this method were called outside of the API Gateway console.

> **Note**
>
> Although the CloudWatch Logs entries are simulated, the results of the method call are real.

In addition to using the API Gateway console, you can use AWS CLI or an AWS SDK for API Gateway to test invoking a method. To do so using AWS CLI, see test-invoke-method.

Call REST API through generated SDKs

This section shows how to call an API through a generated SDK in a client app written in Java, Java for Android, JavaScript, Ruby, Objective-C and Swift.

Topics

Use a Java SDK generated by API Gateway for a REST API

In this section, we outline the steps to use a Java SDK generated by API Gateway for a REST API, by using the Simple Calculator API as an example. Before proceeding, you must complete the steps in Generate SDKs for an API using the API Gateway console.

To install and use a Java SDK generated by API Gateway

1. Extract the contents of the API Gateway-generated .zip file that you downloaded earlier.
2. Download and install Apache Maven (must be version 3.5 or later).
3. Download and install JDK 8.
4. Set the `JAVA_HOME` environment variable.
5. Go to the unzipped SDK folder where the pom.xml file is located. This folder is `generated-code` by default. Run the **mvn install** command to install the compiled artifact files to your local Maven repository. This creates a `target` folder containing the compiled SDK library.

6. Type the following command in an empty directory to create a client project stub to call the API using the installed SDK library.

```
mvn -B archetype:generate \
    -DarchetypeGroupdId=org.apache.maven.archetypes \
    -DgroupId=examples.aws.apig.simpleCalc.sdk.app \
    -DartifactId=SimpleCalc-sdkClient
```

   Note

   The separator \ in the preceding command is included for readability. The whole command should be on a single line without the separator.

   This command creates an application stub. The application stub contains a `pom.xml` file and an `src` folder under the project's root directory (*`SimpleCalc-sdkClient`* in the preceding command). Initially, there are two source files: `src/main/java/`*`{package-path}`*`/App.java` and `src/test/java/`*`{package-path}`*`/AppTest.java`. In this example, *`{package-path}`* is `examples/aws/apig/simpleCalc/sdk/app`. This package path is derived from the `DarchetypeGroupdId` value. You can use the `App.java` file as a template for your client application, and you can add others in the same folder if needed. You can use the `AppTest.java` file as a unit test template for your application, and you can add other test code files to the same test folder as needed.

7. Update the package dependencies in the generated `pom.xml` file to the following, substituting your project's `groupId`, `artifactId`, `version`, and `name` properties, if necessary:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>examples.aws.apig.simpleCalc.sdk.app</groupId>
  <artifactId>SimpleCalc-sdkClient</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleCalc-sdkClient</name>
  <url>http://maven.apache.org</url>

   <dependencies>
      <dependency>
```

```
          <groupId>com.amazonaws</groupId>
          <artifactId>aws-java-sdk-core</artifactId>
          <version>1.11.94</version>
      </dependency>
      <dependency>
          <groupId>my-apig-api-examples</groupId>
          <artifactId>simple-calc-sdk</artifactId>
          <version>1.0.0</version>
      </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.5</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

> **Note**
>
> When a newer version of dependent artifact of `aws-java-sdk-core` is incompatible with the version specified above (`1.11.94`), you must update the `<version>` tag to the new version.

8. Next, we show how to call the API using the SDK by calling the `getABOp(GetABOpRequest req)`, `getApiRoot(GetApiRootRequest req)`, and `postApiRoot(PostApiRootRequest req)` methods of the SDK. These methods correspond to the `GET /{a}/{b}/{op}`, `GET /?a={x}&b={y}&op={operator}`, and `POST /` methods, with a payload of `{"a": x, "b": y, "op": "operator"}` API requests, respectively.

   Update the `App.java` file as follows:

```
package examples.aws.apig.simpleCalc.sdk.app;

import java.io.IOException;

import com.amazonaws.opensdk.config.ConnectionConfiguration;
import com.amazonaws.opensdk.config.TimeoutConfiguration;

import examples.aws.apig.simpleCalc.sdk.*;
import examples.aws.apig.simpleCalc.sdk.model.*;
import examples.aws.apig.simpleCalc.sdk.SimpleCalcSdk.*;

public class App
{
    SimpleCalcSdk sdkClient;

    public App() {
        initSdk();
    }

    // The configuration settings are for illustration purposes and may not be a
 recommended best practice.
    private void initSdk() {
        sdkClient = SimpleCalcSdk.builder()
              .connectionConfiguration(
                  new ConnectionConfiguration()
                        .maxConnections(100)
```

```
                       .connectionMaxIdleMillis(1000))
             .timeoutConfiguration(
                 new TimeoutConfiguration()
                       .httpRequestTimeout(3000)
                       .totalExecutionTimeout(10000)
                       .socketTimeout(2000))
       .build();

   }
   // Calling shutdown is not necessary unless you want to exert explicit control
of this resource.
   public void shutdown() {
       sdkClient.shutdown();
   }

   // GetABOpResult getABOp(GetABOpRequest getABOpRequest)
   public Output getResultWithPathParameters(String x, String y, String operator)
{
    operator = operator.equals("+") ? "add" : operator;
    operator = operator.equals("/") ? "div" : operator;

       GetABOpResult abopResult = sdkClient.getABOp(new
GetABOpRequest().a(x).b(y).op(operator));
       return abopResult.getResult().getOutput();
   }

   public Output getResultWithQueryParameters(String a, String b, String op) {
       GetApiRootResult rootResult = sdkClient.getApiRoot(new
GetApiRootRequest().a(a).b(b).op(op));
       return rootResult.getResult().getOutput();
   }

   public Output getResultByPostInputBody(Double x, Double y, String o) {
    PostApiRootResult postResult = sdkClient.postApiRoot(
     new PostApiRootRequest().input(new Input().a(x).b(y).op(o)));
    return postResult.getResult().getOutput();
   }

   public static void main( String[] args )
   {
       System.out.println( "Simple calc" );
       // to begin
       App calc = new App();
```

```
        // call the SimpleCalc API
        Output res = calc.getResultWithPathParameters("1", "2", "-");
        System.out.printf("GET /1/2/-: %s\n", res.getC());

        // Use the type query parameter
        res = calc.getResultWithQueryParameters("1", "2", "+");
        System.out.printf("GET /?a=1&b=2&op=+: %s\n", res.getC());

        // Call POST with an Input body.
        res = calc.getResultByPostInputBody(1.0, 2.0, "*");
        System.out.printf("PUT /\n\n{\"a\":1, \"b\":2,\"op\":\"*\"}\n %s\n",
 res.getC());


    }
}
```

In the preceding example, the configuration settings used to instantiate the SDK client are for illustration purposes and are not necessarily recommended best practice. Also, calling `sdkClient.shutdown()` is optional, especially if you need precise control on when to free up resources.

We have shown the essential patterns to call an API using a Java SDK. You can extend the instructions to calling other API methods.

Use an Android SDK generated by API Gateway for a REST API

In this section, we will outline the steps to use an Android SDK generated by API Gateway for a REST API. Before proceeding further, you must have already completed the steps in Generate SDKs for an API using the API Gateway console.

> **Note**
>
> The generated SDK is not compatible with Android 4.4 and earlier. For more information, see the section called “Important notes”.

To install and use an Android SDK generated by API Gateway

1. Extract the contents of the API Gateway-generated .zip file that you downloaded earlier.

2. Download and install Apache Maven (preferably version 3.x).
3. Download and install JDK 8.
4. Set the `JAVA_HOME` environment variable.
5. Run the **mvn install** command to install the compiled artifact files to your local Maven repository. This creates a `target` folder containing the compiled SDK library.
6. Copy the SDK file (the name of which is derived from the **Artifact Id** and **Artifact Version** you specified when generating the SDK, e.g., `simple-calcsdk-1.0.0.jar`) from the `target` folder, along with all of the other libraries from the `target/lib` folder, into your project's `lib` folder.

   If you use Android Studio, create a `libs` folder under your client app module and copy the required .jar file into this folder. Verify that the dependencies section in the module's gradle file contains the following.

   ```
       compile fileTree(include: ['*.jar'], dir: 'libs')
       compile fileTree(include: ['*.jar'], dir: 'app/libs')
   ```

   Make sure no duplicated .jar files are declared.
7. Use the `ApiClientFactory` class to initialize the API Gateway-generated SDK. For example:

   ```
   ApiClientFactory factory = new ApiClientFactory();

   // Create an instance of your SDK. Here, 'SimpleCalcClient.java' is the compiled
    java class for the SDK generated by API Gateway.
   final SimpleCalcClient client = factory.build(SimpleCalcClient.class);

   // Invoke a method:
   //   For the 'GET /?a=1&b=2&op=+' method exposed by the API, you can invoke it by
    calling the following SDK method:

   Result output = client.rootGet("1", "2", "+");

   //     where the Result class of the SDK corresponds to the Result model of the
    API.
   //

   //   For the 'GET /{a}/{b}/{op}'  method exposed by the API, you can call the
    following SDK method to invoke the request,
   ```

```
Result output = client.aBOpGet(a, b, c);

//     where a, b, c can be "1", "2", "add", respectively.

//   For the following API method:
//        POST /
//        host: ...
//        Content-Type: application/json
//
//        { "a": 1, "b": 2, "op": "+" }
// you can call invoke it by calling the rootPost method of the SDK as follows:
Input body = new Input();
input.a=1;
input.b=2;
input.op="+";
Result output = client.rootPost(body);

//      where the Input class of the SDK corresponds to the Input model of the API.

// Parse the result:
//     If the 'Result' object is { "a": 1, "b": 2, "op": "add", "c":3"}, you
 retrieve the result 'c') as

String result=output.c;
```

8. To use an Amazon Cognito credentials provider to authorize calls to your API, use the `ApiClientFactory` class to pass a set of AWS credentials by using the SDK generated by API Gateway, as shown in the following example.

```
// Use CognitoCachingCredentialsProvider to provide AWS credentials
// for the ApiClientFactory
AWSCredentialsProvider credentialsProvider = new CognitoCachingCredentialsProvider(
        context,          // activity context
        "identityPoolId", // Cognito identity pool id
        Regions.US_EAST_1 // region of Cognito identity pool
);

ApiClientFactory factory = new ApiClientFactory()
  .credentialsProvider(credentialsProvider);
```

9. To set an API key by using the API Gateway- generated SDK, use code similar to the following.

```
ApiClientFactory factory = new ApiClientFactory()
  .apiKey("YOUR_API_KEY");
```

Use a JavaScript SDK generated by API Gateway for a REST API

Note

These instructions assume you have already completed the instructions in Generate SDKs for an API using the API Gateway console.

Important

If your API only has ANY methods defined, the generated SDK package will not contain an `apigClient.js` file, and you will need to define the ANY methods yourself.

To install, initiate and call a JavaScript SDK generated by API Gateway for a REST API

1. Extract the contents of the API Gateway-generated .zip file you downloaded earlier.
2. Enable cross-origin resource sharing (CORS) for all of the methods the SDK generated by API Gateway will call. For instructions, see Enabling CORS for a REST API resource.
3. In your web page, include references to the following scripts.

```
<script type="text/javascript" src="lib/axios/dist/axios.standalone.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/hmac-sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/hmac.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/enc-base64.js"></
script>
<script type="text/javascript" src="lib/url-template/url-template.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/sigV4Client.js"></script>
```

```
<script type="text/javascript" src="lib/apiGatewayCore/apiGatewayClient.js"></
script>
<script type="text/javascript" src="lib/apiGatewayCore/simpleHttpClient.js"></
script>
<script type="text/javascript" src="lib/apiGatewayCore/utils.js"></script>
<script type="text/javascript" src="apigClient.js"></script>
```

4. In your code, initialize the SDK generated by API Gateway by using code similar to the following.

   ```
   var apigClient = apigClientFactory.newClient();
   ```

   To initialize the SDK generated by API Gateway with AWS credentials, use code similar to the following. If you use AWS credentials, all requests to the API will be signed.

   ```
   var apigClient = apigClientFactory.newClient({
     accessKey: 'ACCESS_KEY',
     secretKey: 'SECRET_KEY',
   });
   ```

   To use an API key with the SDK generated by API Gateway, pass the API key as a parameter to the `Factory` object by using code similar to the following. If you use an API key, it is specified as part of the `x-api-key` header and all requests to the API will be signed. This means you must set the appropriate CORS Accept headers for each request.

   ```
   var apigClient = apigClientFactory.newClient({
     apiKey: 'API_KEY'
   });
   ```

5. Call the API methods in API Gateway by using code similar to the following. Each call returns a promise with a success and failure callbacks.

   ```
   var params = {
     // This is where any modeled request parameters should be added.
     // The key is the parameter name, as it is defined in the API in API Gateway.
     param0: '',
     param1: ''
   };

   var body = {
   ```

```
  // This is where you define the body of the request,
};

var additionalParams = {
  // If there are any unmodeled query parameters or headers that must be
  //   sent with the request, add them here.
  headers: {
    param0: '',
    param1: ''
  },
  queryParams: {
    param0: '',
    param1: ''
  }
};

apigClient.methodName(params, body, additionalParams)
    .then(function(result){
      // Add success callback code here.
    }).catch( function(result){
      // Add error callback code here.
    });
```

Here, the *methodName* is constructed from the method request's resource path and the HTTP verb. For the SimpleCalc API, the SDK methods for the API methods of

```
1. GET /?a=...&b=...&op=...
2. POST /

   { "a": ..., "b": ..., "op": ...}
3. GET /{a}/{b}/{op}
```

the corresponding SDK methods are as follows:

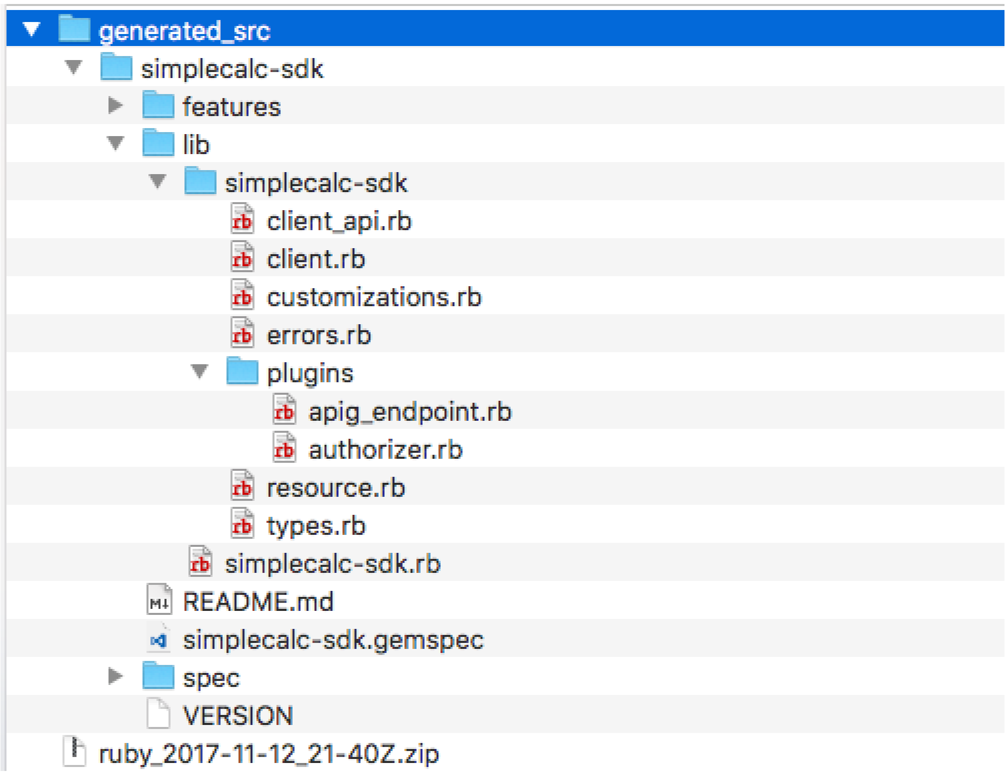
```
1. rootGet(params);      // where params={"a": ..., "b": ..., "op": ...} is
 resolved to the query parameters
2. rootPost(null, body); // where body={"a": ..., "b": ..., "op": ...}
3. aBOpGet(params);      // where params={"a": ..., "b": ..., "op": ...} is
 resolved to the path parameters
```

Use a Ruby SDK generated by API Gateway for a REST API

> **Note**
>
> These instructions assume you already completed the instructions in Generate SDKs for an API using the API Gateway console.

To install, instantiate, and call a Ruby SDK generated by API Gateway for a REST API

1. Unzip the downloaded Ruby SDK file. The generated SDK source is shown as follows.



```
▼ generated_src
    ▼ simplecalc-sdk
        ▶ features
        ▼ lib
            ▼ simplecalc-sdk
                  client_api.rb
                  client.rb
                  customizations.rb
                  errors.rb
                ▼ plugins
                      apig_endpoint.rb
                      authorizer.rb
                  resource.rb
                  types.rb
              simplecalc-sdk.rb
          README.md
          simplecalc-sdk.gemspec
        ▶ spec
          VERSION
    ruby_2017-11-12_21-40Z.zip
```

2. Build a Ruby Gem from the generated SDK source, using the following shell commands in a terminal window:

```
# change to /simplecalc-sdk directory
cd simplecalc-sdk

# build the generated gem
gem build simplecalc-sdk.gemspec
```

After this, **simplecalc-sdk-1.0.0.gem** becomes available.

3. Install the gem:

```
gem install simplecalc-sdk-1.0.0.gem
```

4. Create a client application. Instantiate and initialize the Ruby SDK client in the app:

```
require 'simplecalc-sdk'
client = SimpleCalc::Client.new(
    http_wire_trace: true,
    retry_limit: 5,
    http_read_timeout: 50
)
```

If the API has authorization of the `AWS_IAM` type is configured, you can include the caller's AWS credentials by supplying `accessKey` and `secretKey` during the initialization:

```
require 'pet-sdk'
client = Pet::Client.new(
    http_wire_trace: true,
    retry_limit: 5,
    http_read_timeout: 50,
    access_key_id: 'ACCESS_KEY',
    secret_access_key: 'SECRET_KEY'
)
```

5. Make API calls through the SDK in the app.

> **Tip**
>
> If you are not familiar with the SDK method call conventions, you can review the `client.rb` file in the generated SDK `lib` folder. The folder contains documentation of each supported API method call.

To discover supported operations:

```
# to show supported operations:
puts client.operation_names
```

This results in the following display, corresponding to the API methods of `GET /?a={.}&b={.}&op={.}`, `GET /{a}/{b}/{op}`, and `POST /`, plus a payload of the `{a:"…", b:"…", op:"…"}` format, respectively:

```
[:get_api_root, :get_ab_op, :post_api_root]
```

To invoke the `GET /?a=1&b=2&op=+` API method, call the following the Ruby SDK method:

```
resp = client.get_api_root({a:"1", b:"2", op:"+"})
```

To invoke the `POST /` API method with a payload of `{a: "1", b: "2", "op": "+"}`, call the following Ruby SDK method:

```
resp = client.post_api_root(input: {a:"1", b:"2", op:"+"})
```

To invoke the `GET /1/2/+` API method, call the following Ruby SDK method:

```
resp = client.get_ab_op({a:"1", b:"2", op:"+"})
```

The successful SDK method calls return the following response:

```
resp : {
    result: {
        input: {
            a: 1,
            b: 2,
            op: "+"
        },
        output: {
            c: 3
        }
    }
}
```

Use iOS SDK generated by API Gateway for a REST API in Objective-C or Swift

In this tutorial, we will show how to use an iOS SDK generated by API Gateway for a REST API in an Objective-C or Swift app to call the underlying API. We will use the SimpleCalc API as an example to illustrate the following topics:

- How to install the required AWS Mobile SDK components into your Xcode project
- How to create the API client object before calling the API's methods
- How to call the API methods through the corresponding SDK methods on the API client object
- How to prepare a method input and parse its result using the corresponding model classes of the SDK

Topics

Use generated iOS SDK (Objective-C) to call API

Before beginning the following procedure, you must complete the steps in Generate SDKs for an API using the API Gateway console for iOS in Objective-C and download the .zip file of the generated SDK.

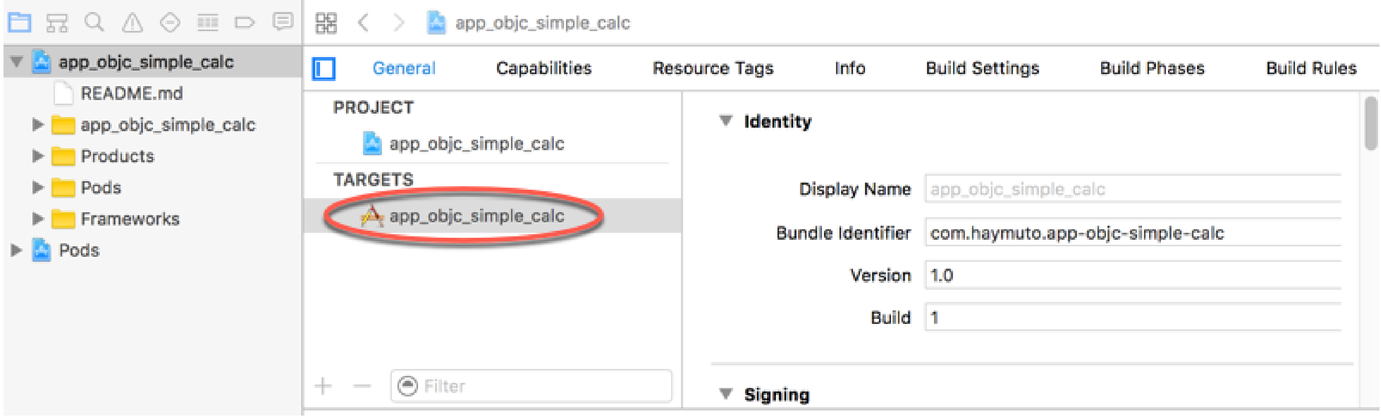
Install the AWS mobile SDK and an iOS SDK generated by API Gateway in an Objective-C project

The following procedure describes how to install the SDK.

To install and use an iOS SDK generated by API Gateway in Objective-C

1. Extract the contents of the API Gateway-generated .zip file you downloaded earlier. Using the SimpleCalc API, you may want to rename the unzipped SDK folder to something like **`sdk_objc_simple_calc`**. In this SDK folder there is a `README.md` file and a `Podfile` file. The `README.md` file contains the instructions to install and use the SDK. This tutorial provides details about these instructions. The installation leverages CocoaPods to import required API Gateway libraries and other dependent AWS Mobile SDK components. You must update the `Podfile` to import the SDKs into your app's Xcode project. The unarchived SDK folder also contains a `generated-src` folder that contains the source code of the generated SDK of your API.

2. Launch Xcode and create a new iOS Objective-C project. Make a note of the project's target. You will need to set it in the Podfile.



Display Name: app_objc_simple_calc
Bundle Identifier: com.haymuto.app-objc-simple-calc
Version: 1.0
Build: 1

3. To import the AWS Mobile SDK for iOS into the Xcode project by using CocoaPods, do the following:

   a. Install CocoaPods by running the following command in a terminal window:

   ```
   sudo gem install cocoapods
   pod setup
   ```

   b. Copy the Podfile file from the extracted SDK folder into the same directory containing your Xcode project file. Replace the following block:

   ```
   target '<YourXcodeTarget>' do
       pod 'AWSAPIGateway', '~> 2.4.7'
   end
   ```

   with your project's target name:

   ```
   target 'app_objc_simple_calc' do
       pod 'AWSAPIGateway', '~> 2.4.7'
   end
   ```

   If your Xcode project already contains a file named Podfile, add the following line of code to it:

   ```
   pod 'AWSAPIGateway', '~> 2.4.7'
   ```

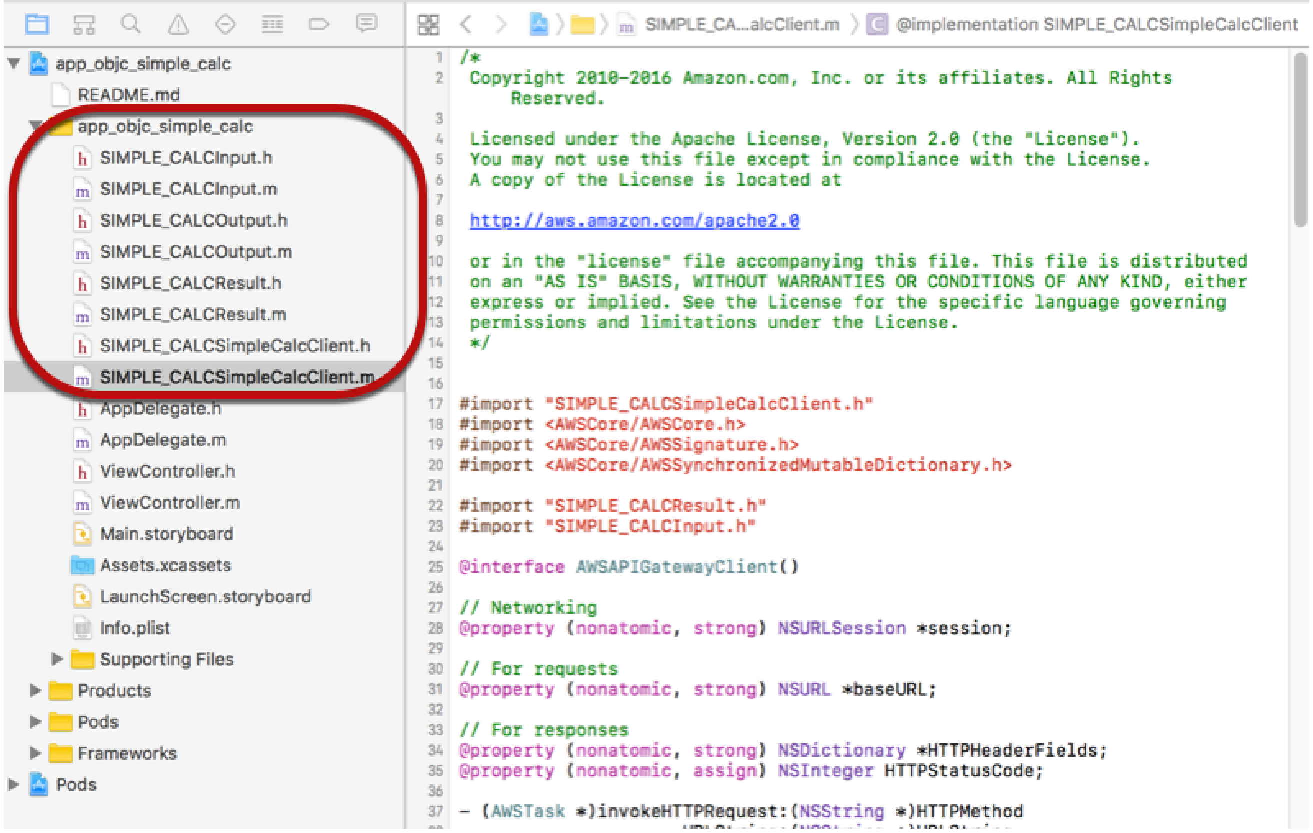
   c. Open a terminal window and run the following command:

```
pod install
```

This installs the API Gateway component and other dependent AWS Mobile SDK components.

d. Close the Xcode project and then open the `.xcworkspace` file to relaunch Xcode.

e. Add all of the `.h` and `.m` files from the extracted SDK's `generated-src` directory into your Xcode project.



```
/*
 Copyright 2010-2016 Amazon.com, Inc. or its affiliates. All Rights
     Reserved.

 Licensed under the Apache License, Version 2.0 (the "License").
 You may not use this file except in compliance with the License.
 A copy of the License is located at

 http://aws.amazon.com/apache2.0

 or in the "license" file accompanying this file. This file is distributed
 on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 express or implied. See the License for the specific language governing
 permissions and limitations under the License.
 */


#import "SIMPLE_CALCSimpleCalcClient.h"
#import <AWSCore/AWSCore.h>
#import <AWSCore/AWSSignature.h>
#import <AWSCore/AWSSynchronizedMutableDictionary.h>

#import "SIMPLE_CALCResult.h"
#import "SIMPLE_CALCInput.h"

@interface AWSAPIGatewayClient()

// Networking
@property (nonatomic, strong) NSURLSession *session;

// For requests
@property (nonatomic, strong) NSURL *baseURL;

// For responses
@property (nonatomic, strong) NSDictionary *HTTPHeaderFields;
@property (nonatomic, assign) NSInteger HTTPStatusCode;

- (AWSTask *)invokeHTTPRequest:(NSString *)HTTPMethod
```

To import the AWS Mobile SDK for iOS Objective-C into your project by explicitly downloading AWS Mobile SDK or using Carthage, follow the instructions in the *README.md* file. Be sure to use only one of these options to import the AWS Mobile SDK.

Call API methods using the iOS SDK generated by API Gateway in an Objective-C project

When you generated the SDK with the prefix of `SIMPLE_CALC` for this SimpleCalc API with two models for input (`Input`) and output (`Result`) of the methods, in the SDK, the resulting API client class becomes SIMPLE_CALCSimpleCalcClient and the corresponding data classes are SIMPLE_CALCInput and SIMPLE_CALCResult, respectively. The API requests and responses are mapped to the SDK methods as follows:

- The API request of

  ```
  GET /?a=...&b=...&op=...
  ```

  becomes the SDK method of

  ```
  (AWSTask *)rootGet:(NSString *)op a:(NSString *)a b:(NSString *)b
  ```

  The AWSTask.result property is of the SIMPLE_CALCResult type if the Result model was added to the method response. Otherwise, the property is of the NSDictionary type.
- This API request of

  ```
  POST /

  {
     "a": "Number",
     "b": "Number",
     "op": "String"
  }
  ```

  becomes the SDK method of

  ```
  (AWSTask *)rootPost:(SIMPLE_CALCInput *)body
  ```
- The API request of

  ```
  GET /{a}/{b}/{op}
  ```

  becomes the SDK method of

  ```
  (AWSTask *)aBOpGet:(NSString *)a b:(NSString *)b op:(NSString *)op
  ```

The following procedure describes how to call the API methods in Objective-C app source code; for example, as part of the viewDidLoad delegate in a ViewController.m file.

To call the API through the iOS SDK generated by API Gateway

1. Import the API client class header file to make the API client class callable in the app:

   ```
   #import "SIMPLE_CALCSimpleCalc.h"
   ```

   The `#import` statement also imports `SIMPLE_CALCInput.h` and `SIMPLE_CALCResult.h` for the two model classes.

2. Instantiate the API client class:

   ```
   SIMPLE_CALCSimpleCalcClient *apiInstance = [SIMPLE_CALCSimpleCalcClient
    defaultClient];
   ```

   To use Amazon Cognito with the API, set the `defaultServiceConfiguration` property on the default `AWSServiceManager` object, as shown in the following, before calling the `defaultClient` method to create the API client object (shown in the preceding example):

   ```
   AWSCognitoCredentialsProvider *creds = [[AWSCognitoCredentialsProvider alloc]
    initWithRegionType:AWSRegionUSEast1 identityPoolId:your_cognito_pool_id];
   AWSServiceConfiguration *configuration = [[AWSServiceConfiguration alloc]
    initWithRegion:AWSRegionUSEast1 credentialsProvider:creds];
   AWSServiceManager.defaultServiceManager.defaultServiceConfiguration =
    configuration;
   ```

3. Call the `GET /?a=1&b=2&op=+` method to perform `1+2`:

   ```
   [[apiInstance rootGet: @"+" a:@"1" b:@"2"] continueWithBlock:^id _Nullable(AWSTask
    * _Nonnull task) {
       _textField1.text = [self handleApiResponse:task];
       return nil;
   }];
   ```

   where the helper function `handleApiResponse:task` formats the result as a string to be displayed in a text field (`_textField1`).

   ```
   - (NSString *)handleApiResponse:(AWSTask *)task {
       if (task.error != nil) {
           return [NSString stringWithFormat: @"Error: %@", task.error.description];
       } else if (task.result != nil && [task.result isKindOfClass:[SIMPLE_CALCResult
    class]]) {
   ```

```
        return [NSString stringWithFormat:@"%@ %@ %@ = %@\n",task.result.input.a,
 task.result.input.op, task.result.input.b, task.result.output.c];
    }
    return nil;
}
```

The resulting display is `1 + 2 = 3`.

4. Call the `POST /` with a payload to perform `1-2`:

   ```
   SIMPLE_CALCInput *input = [[SIMPLE_CALCInput alloc] init];
       input.a = [NSNumber numberWithInt:1];
       input.b = [NSNumber numberWithInt:2];
       input.op = @"-";
       [[apiInstance rootPost:input] continueWithBlock:^id _Nullable(AWSTask *
    _Nonnull task) {
           _textField2.text = [self handleApiResponse:task];
           return nil;
       }];
   ```

   The resulting display is `1 - 2 = -1`.

5. Call the `GET /{a}/{b}/{op}` to perform `1/2`:

   ```
   [[apiInstance aBOpGet:@"1" b:@"2" op:@"div"] continueWithBlock:^id
    _Nullable(AWSTask * _Nonnull task) {
       _textField3.text = [self handleApiResponse:task];
       return nil;
   }];
   ```

   The resulting display is `1 div 2 = 0.5`. Here, `div` is used in place of `/` because the simple Lambda function in the backend does not handle URL encoded path variables.

Use generated iOS SDK (Swift) to call API

Before beginning the following procedure, you must complete the steps in Generate SDKs for an API using the API Gateway console for iOS in Swift and download the .zip file of the generated SDK.

Topics

- Install AWS mobile SDK and API Gateway-generated SDK in a Swift project

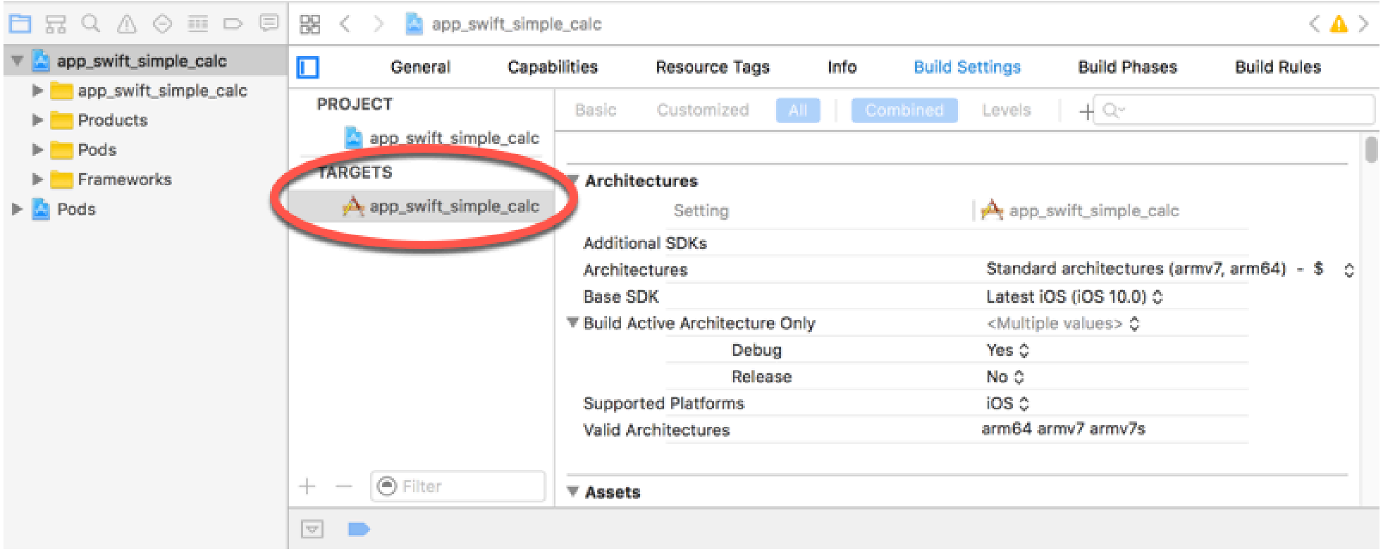
- Call API methods through the iOS SDK generated by API Gateway in a Swift project

Install AWS mobile SDK and API Gateway-generated SDK in a Swift project

The following procedure describes how to install the SDK.

To install and use an iOS SDK generated by API Gateway in Swift

1. Extract the contents of the API Gateway-generated .zip file you downloaded earlier. Using the SimpleCalc API, you may want to rename the unzipped SDK folder to something like **`sdk_swift_simple_calc`**. In this SDK folder there is a `README.md` file and a `Podfile` file. The `README.md` file contains the instructions to install and use the SDK. This tutorial provides details about these instructions. The installation leverages CocoaPods to import required AWS Mobile SDK components. You must update the `Podfile` to import the SDKs into your Swift app's Xcode project. The unarchived SDK folder also contains a `generated-src` folder that contains the source code of the generated SDK of your API.
2. Launch Xcode and create a new iOS Swift project. Make a note of the project's target. You will need to set it in the `Podfile`.



Architectures

Setting	app_swift_simple_calc
Additional SDKs	
Architectures	Standard architectures (armv7, arm64) - $ ◇
Base SDK	Latest iOS (iOS 10.0) ◇
Build Active Architecture Only	<Multiple values> ◇
Debug	Yes ◇
Release	No ◇
Supported Platforms	iOS ◇
Valid Architectures	arm64 armv7 armv7s

3. To import the required AWS Mobile SDK components into the Xcode project by using CocoaPods, do the following:

   a. If it is not installed, install CocoaPods by running the following command in a terminal window:

   ```
   sudo gem install cocoapods
   ```

```
pod setup
```

b. Copy the Podfile file from the extracted SDK folder into the same directory containing your Xcode project file. Replace the following block:

```
target '<YourXcodeTarget>' do
    pod 'AWSAPIGateway', '~> 2.4.7'
end
```

with your project's target name as shown:

```
target 'app_swift_simple_calc' do
    pod 'AWSAPIGateway', '~> 2.4.7'
end
```

If your Xcode project already contains a Podfile with the correct target, you can simply add the following line of code to the do ... end loop:

```
pod 'AWSAPIGateway', '~> 2.4.7'
```

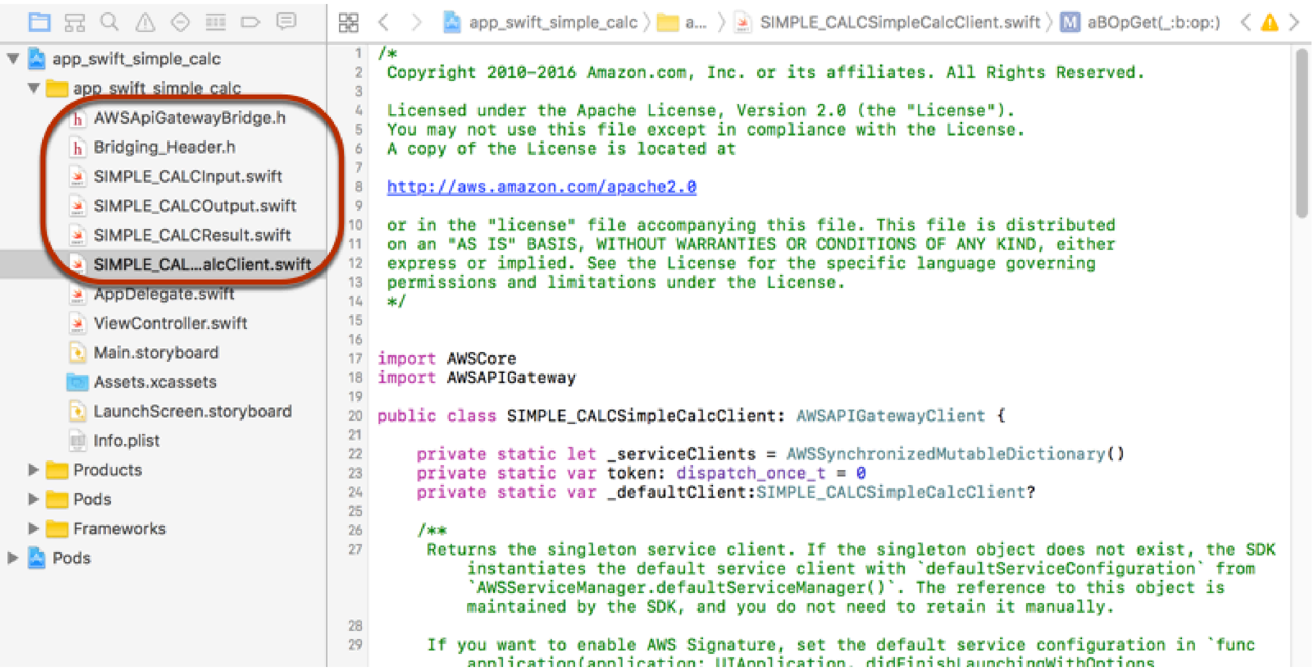
c. Open a terminal window and run the following command in the app directory:

```
pod install
```

This installs the API Gateway component and any dependent AWS Mobile SDK components into the app's project.

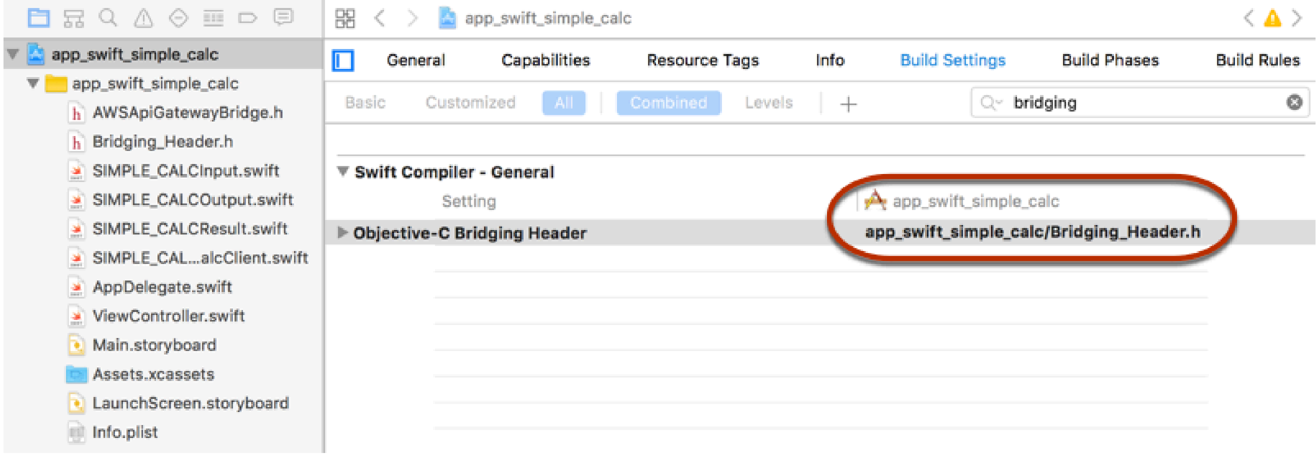
d. Close the Xcode project and then open the *.xcworkspace file to relaunch Xcode.

e. Add all of the SDK's header files (.h) and Swift source code files (.swift) from the extracted generated-src directory to your Xcode project.



```
/*
 Copyright 2010-2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.

 Licensed under the Apache License, Version 2.0 (the "License").
 You may not use this file except in compliance with the License.
 A copy of the License is located at

 http://aws.amazon.com/apache2.0

 or in the "license" file accompanying this file. This file is distributed
 on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 express or implied. See the License for the specific language governing
 permissions and limitations under the License.
 */


import AWSCore
import AWSAPIGateway

public class SIMPLE_CALCSimpleCalcClient: AWSAPIGatewayClient {

    private static let _serviceClients = AWSSynchronizedMutableDictionary()
    private static var token: dispatch_once_t = 0
    private static var _defaultClient:SIMPLE_CALCSimpleCalcClient?

    /**
     Returns the singleton service client. If the singleton object does not exist, the SDK
         instantiates the default service client with `defaultServiceConfiguration` from
          `AWSServiceManager.defaultServiceManager()`. The reference to this object is
         maintained by the SDK, and you do not need to retain it manually.

     If you want to enable AWS Signature, set the default service configuration in `func
         application(application: UIApplication, didFinishLaunchingWithOptions
```

f. To enable calling the Objective-C libraries of the AWS Mobile SDK from your Swift code project, set the `Bridging_Header.h` file path on the **Objective-C Bridging Header** property under the **Swift Compiler - General** setting of your Xcode project configuration:



bridging

Swift Compiler - General

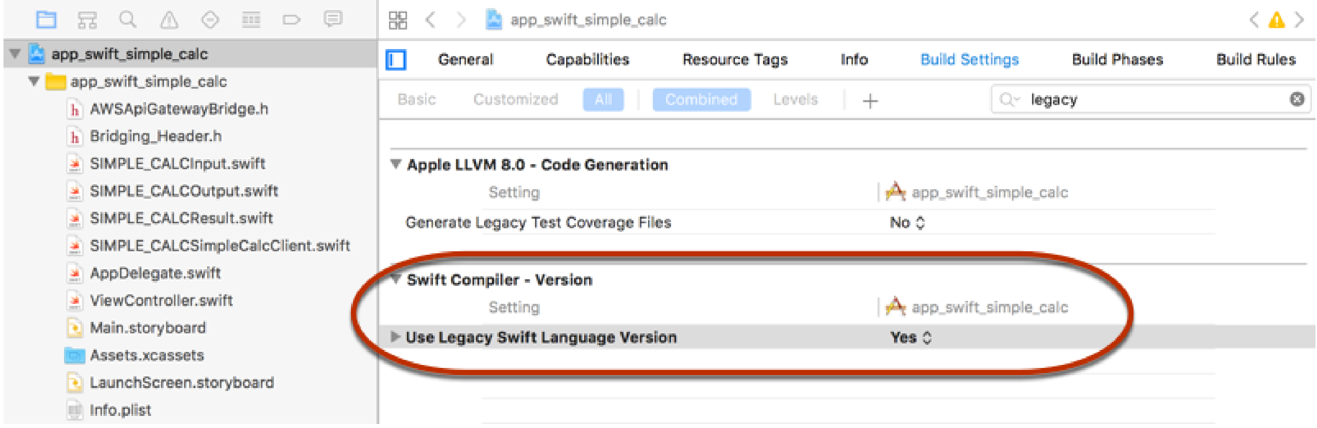
Setting	app_swift_simple_calc
Objective-C Bridging Header	app_swift_simple_calc/Bridging_Header.h

Tip

You can type **`bridging`** in the search box of Xcode to locate the **Objective-C Bridging Header** property.

g. Build the Xcode project to verify that it is properly configured before proceeding further. If your Xcode uses a more recent version of Swift than the one supported for the AWS

Mobile SDK, you will get Swift compiler errors. In this case, set the **Use Legacy Swift Language Version** property to **Yes** under the **Swift Compiler - Version** setting:



Apple LLVM 8.0 - Code Generation

Setting	app_swift_simple_calc
Generate Legacy Test Coverage Files	No

Swift Compiler - Version

Setting	app_swift_simple_calc
Use Legacy Swift Language Version	Yes

To import the AWS Mobile SDK for iOS in Swift into your project by explicitly downloading the AWS Mobile SDK or using Carthage, follow the instructions in the README.md file that comes with the SDK package. Be sure to use only one of these options to import the AWS Mobile SDK.

Call API methods through the iOS SDK generated by API Gateway in a Swift project

When you generated the SDK with the prefix of SIMPLE_CALC for this SimpleCalc API with two models to describe the input (Input) and output (Result) of the API's requests and responses, in the SDK, the resulting API client class becomes SIMPLE_CALCSimpleCalcClient and the corresponding data classes are SIMPLE_CALCInput and SIMPLE_CALCResult, respectively. The API requests and responses are mapped to the SDK methods as follows:

- The API request of

  ```
  GET /?a=...&b=...&op=...
  ```

  becomes the SDK method of

  ```
  public func rootGet(op: String?, a: String?, b: String?) -> AWSTask
  ```

  The AWSTask.result property is of the SIMPLE_CALCResult type if the Result model was added to the method response. Otherwise, it is of the NSDictionary type.

- This API request of

```
POST /

{
   "a": "Number",
   "b": "Number",
   "op": "String"
}
```

becomes the SDK method of

```
public func rootPost(body: SIMPLE_CALCInput) -> AWSTask
```

- The API request of

```
GET /{a}/{b}/{op}
```

becomes the SDK method of

```
public func aBOpGet(a: String, b: String, op: String) -> AWSTask
```

The following procedure describes how to call the API methods in Swift app source code; for example, as part of the `viewDidLoad()` delegate in a `ViewController.m` file.

To call the API through the iOS SDK generated by API Gateway

1. Instantiate the API client class:

   ```
   let client = SIMPLE_CALCSimpleCalcClient.default()
   ```

   To use Amazon Cognito with the API, set a default AWS service configuration (shown following) before getting the `default` method (shown previously):

   ```
   let credentialsProvider =
    AWSCognitoCredentialsProvider(regionType: AWSRegionType.USEast1, identityPoolId:
    "my_pool_id")
   let configuration = AWSServiceConfiguration(region: AWSRegionType.USEast1,
    credentialsProvider: credentialsProvider)
   ```

```
AWSServiceManager.defaultServiceManager().defaultServiceConfiguration =
 configuration
```

2. Call the GET `/?a=1&b=2&op=+` method to perform `1+2`:

   ```
   client.rootGet("+", a: "1", b:"2").continueWithBlock {(task: AWSTask) -> AnyObject?
    in
       self.showResult(task)
       return nil
   }
   ```

   where the helper function `self.showResult(task)` prints the result or error to the console; for example:

   ```
   func showResult(task: AWSTask) {
       if let error = task.error {
           print("Error: \(error)")
       } else if let result = task.result {
           if result is SIMPLE_CALCResult {
               let res = result as! SIMPLE_CALCResult
               print(String(format:"%@ %@ %@ = %@", res.input!.a!, res.input!.op!,
    res.input!.b!, res.output!.c!))
           } else if result is NSDictionary {
               let res = result as! NSDictionary
               print("NSDictionary: \(res)")
           }
       }
   }
   ```

   In a production app, you can display the result or error in a text field. The resulting display is `1 + 2 = 3`.

3. Call the `POST /` with a payload to perform `1-2`:

   ```
   let body = SIMPLE_CALCInput()
   body.a=1
   body.b=2
   body.op="-"
   client.rootPost(body).continueWithBlock {(task: AWSTask) -> AnyObject? in
       self.showResult(task)
       return nil
   }
   ```

The resultant display is `1 - 2 = -1`.

4. Call the GET `/{a}/{b}/{op}` to perform 1/2:

```
client.aBOpGet("1", b:"2", op:"div").continueWithBlock {(task: AWSTask) ->
 AnyObject? in
    self.showResult(task)
    return nil
}
```

The resulting display is `1 div 2 = 0.5`. Here, `div` is used in place of / because the simple Lambda function in the backend does not handle URL encoded path variables.

How to invoke a private API

Private APIs are accessible only from within your VPCs, and the resource policies must allow access from the VPCs and VPC endpoints you have configured. How you access your private API will depend upon whether or not you have enabled private DNS on the VPC endpoint. For example, while accessing private API from on-premises network via AWS Direct Connect, you will have private DNS enabled on the VPC endpoint. In such a case, follow the steps outlined in Invoking Your Private API Using Endpoint-Specific Public DNS Hostnames.

Once you have deployed a private API, you can access it via private DNS (if you've enabled private DNS naming) and via public DNS.

To get the DNS names for your private API, do the following:

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the left navigation pane, choose **Endpoints** and then choose your interface VPC endpoint for API Gateway.
3. In the **Details** pane, you'll see 5 values in the **DNS names** field. The first 3 are the public DNS names for your API. The other 2 are the private DNS names for it.

Invoking your private API using private DNS names

> ⚠ **Warning**
>
> When you select the Enable Private DNS Name option while creating an interface VPC endpoint for API Gateway, the VPC where the VPC Endpoint is present won't be able to access public (edge-optimized and regional) APIs. For more information, see Why can't I connect to my public API from an API Gateway VPC endpoint?.

If you've enabled private DNS, you can access your private API using the private DNS names as follows:

```
{restapi-id}.execute-api.{region}.amazonaws.com
```

The base URL to invoke the API is in the following format:

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stage}
```

For example, assuming you set up the `GET /pets` and `GET /pets/{petId}` methods in this example, and assuming that your rest API ID was `01234567ab` and your region was `us-west-2`, you could test your API by typing the following URLs in a browser:

```
https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets
```

and

```
https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets/1
```

Alternatively, you could use the following cURL commands:

```
curl -X GET https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets
```

and

```
curl -X GET https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets/2
```

Accessing your private API using AWS Direct Connect

You can also use AWS Direct Connect to establish a dedicated private connection from an on-premises network to Amazon VPC and access your private API endpoint over that connection by using public DNS names.

You can also use private DNS names to access your private API from an on-premises network by setting up an Amazon Route 53 Resolver inbound endpoint and forwarding it all DNS queries of the private DNS from your remote network. For more information, see Forwarding inbound DNS queries to your VPCs in the *Amazon Route 53 Developer Guide*.

Accessing your private API using a Route53 alias

You can associate or disassociate a VPC endpoint with your private API by using the procedure outlined in Associate or Disassociate a VPC Endpoint with a Private REST API.

Once you associate your private API's REST API ID with the VPC endpoints you'll be calling your REST API from, you can use the following format base URL to invoke the API using a Route53 alias.

The generated base URL is in the following format:

```
https://{rest-api-id}-{vpce-id}.execute-api.{region}.amazonaws.com/{stage}
```

For example, assuming you set up the GET /pets and GET /pets/*`{petId}`* methods in this example, and assuming that your API's API ID was `01234567ab`, VPC Endpoint ID was `vpce-01234567abcdef012`, and your Region was `us-west-2`, you can invoke your API as:

```
curl -v https://01234567ab-vpce-01234567abcdef012.execute-api.us-west-2.amazonaws.com/
test/pets/
```

Invoking your private API using endpoint-specific public DNS hostnames

You can access your private API using endpoint-specific DNS hostnames. These are public DNS hostnames containing the VPC endpoint ID or API ID for your private API.

The generated base URL is in the following format:

```
https://{public-dns-hostname}.execute-api.{region}.vpce.amazonaws.com/{stage}
```

For example, assuming you set up the `GET /pets` and `GET /pets/{petId}` methods in this example, and assuming that your API's API ID was `abc1234`, its public DNS hostname was `vpce-`

def-01234567, and your Region was us-west-2, you could test your API via its VPCE ID by using the Host header in a cURL command, as in the following example:

```
curl -v https://vpce-def-01234567.execute-api.us-west-2.vpce.amazonaws.com/test/pets -H
 'Host: abc1234.execute-api.us-west-2.amazonaws.com'
```

Alternatively, you can access your private API via its API ID by using the x-apigw-api-id header in a cURL command in the following format:

```
curl -v https://{public-dns-hostname}.execute-api.{region}.vpce.amazonaws.com/test -
H'x-apigw-api-id:{api-id}'
```

Configuring a REST API using OpenAPI

You can use API Gateway to import a REST API from an external definition file into API Gateway. Currently, API Gateway supports OpenAPI v2.0 and OpenAPI v3.0 definition files, with exceptions listed in Amazon API Gateway important notes for REST APIs. You can update an API by overwriting it with a new definition, or you can merge a definition with an existing API. You specify the options by using a mode query parameter in the request URL.

For a tutorial on using the Import API feature from the API Gateway console, see Tutorial: Create a REST API by importing an example.

Topics

Import an edge-optimized API into API Gateway

You can import an API's OpenAPI definition file to create a new edge-optimized API by specifying the EDGE endpoint type as an additional input, besides the OpenAPI file, to the import operation. You can do so using the API Gateway console, AWS CLI, or an AWS SDK.

For a tutorial on using the Import API feature from the API Gateway console, see Tutorial: Create a REST API by importing an example.

Topics

- Import an edge-optimized API using the API Gateway console
- Import an edge-optimized API using the AWS CLI

Import an edge-optimized API using the API Gateway console

To import an edge-optimized API using the API Gateway console, do the following:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**.
3. Under **REST API**, choose **Import**.
4. Copy an API's OpenAPI definition and paste it into the code editor, or choose **Choose file** to load an OpenAPI file from a local drive.
5. For **API endpoint type**, select **Edge-optimized**.
6. Choose **Create API** to start importing the OpenAPI definitions.

Import an edge-optimized API using the AWS CLI

To import an API from an OpenAPI definition file to create a new edge-optimized API using the AWS CLI, use the `import-rest-api` command as follows:

```
aws apigateway import-rest-api \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

or with an explicit specification of the `endpointConfigurationTypes` query string parameter to `EDGE`:

```
aws apigateway import-rest-api \
    --parameters endpointConfigurationTypes=EDGE \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

Import a regional API into API Gateway

When importing an API, you can choose the regional endpoint configuration for the API. You can use the API Gateway console, the AWS CLI, or an AWS SDK.

When you export an API, the API endpoint configuration is not included in the exported API definitions.

For a tutorial on using the Import API feature from the API Gateway console, see Tutorial: Create a REST API by importing an example.

Topics

Import a regional API using the API Gateway console

To import an API of a regional endpoint using the API Gateway console, do the following:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**.
3. Under **REST API**, choose **Import**.
4. Copy an API's OpenAPI definition and paste it into the code editor, or choose **Choose file** to load an OpenAPI file from a local drive.
5. For **API endpoint type**, select **Regional**.
6. Choose **Create API** to start importing the OpenAPI definitions.

Import a regional API using the AWS CLI

To import an API from an OpenAPI definition file using the AWS CLI, use the `import-rest-api` command:

```
aws apigateway import-rest-api \
    --parameters endpointConfigurationTypes=REGIONAL \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

Import an OpenAPI file to update an existing API definition

You can import API definitions only to update an existing API, without changing its endpoint configuration, as well as stages and stage variables, or references to API keys.

The import-to-update operation can occur in two modes: merge or overwrite.

When an API (A) is merged into another (B), the resulting API retains the definitions of both A and B if the two APIs do not share any conflicting definitions. When conflicts arise, the method definitions of the merging API (A) overrides the corresponding method definitions of the merged API (B). For example, suppose B has declared the following methods to return 200 and 206 responses:

```
GET /a
POST /a
```

and A declares the following method to return 200 and 400 responses:

```
GET /a
```

When A is merged into B, the resulting API yields the following methods:

```
GET /a
```

which returns 200 and 400 responses, and

```
POST /a
```

which returns 200 and 206 responses.

Merging an API is useful when you have decomposed your external API definitions into multiple, smaller parts and only want to apply changes from one of those parts at a time. For example, this might occur if multiple teams are responsible for different parts of an API and have changes available at different rates. In this mode, items from the existing API that aren't specifically defined in the imported definition are left alone.

When an API (A) overwrites another API (B), the resulting API takes the definitions of the overwriting API (A). Overwriting an API is useful when an external API definition contains the complete definition of an API. In this mode, items from an existing API that aren't specifically defined in the imported definition are deleted.

To merge an API, submit a PUT request to `https://apigateway.<region>.amazonaws.com/restapis/<restapi_id>?mode=merge`. The `restapi_id` path parameter value specifies the API to which the supplied API definition will be merged.

The following code snippet shows an example of the PUT request to merge an OpenAPI API definition in JSON, as the payload, with the specified API already in API Gateway.

```
PUT /restapis/<restapi_id>?mode=merge
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
Content-Length: ...

An OpenAPI API definition in JSON
```

The merging update operation takes two complete API definitions and merges them together. For a small and incremental change, you can use the resource update operation.

To overwrite an API, submit a PUT request to `https://apigateway.<region>.amazonaws.com/restapis/<restapi_id>?mode=overwrite`. The `restapi_id` path parameter specifies the API that will be overwritten with the supplied API definitions.

The following code snippet shows an example of an overwriting request with the payload of a JSON-formatted OpenAPI definition:

```
PUT /restapis/<restapi_id>?mode=overwrite
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
Content-Length: ...

An OpenAPI API definition in JSON
```

When the `mode` query parameter isn't specified, merge is assumed.

> **Note**
>
> The PUT operations are idempotent, but not atomic. That means if a system error occurs part way through processing, the API can end up in a bad state. However, repeating the operation successfully puts the API into the same final state as if the first operation had succeeded.

Set the OpenAPI basePath property

In OpenAPI 2.0, you can use the `basePath` property to provide one or more path parts that precede each path defined in the `paths` property. Because API Gateway has several ways to express a resource's path, the Import API feature provides the following options for interpreting the `basePath` property during import: ignore, prepend, and split.

In OpenAPI 3.0, `basePath` is no longer a top-level property. Instead, API Gateway uses a server variable as a convention. The Import API feature provides the same options for interpreting the base path during import. The base path is identified as follows:

- If the API doesn't contain any `basePath` variables, the Import API feature checks the `server.url` string to see if it contains a path beyond `"/"`. If it does, that path is used as the base path.
- If the API contains only one `basePath` variable, the Import API feature uses it as the base path, even if it's not referenced in the `server.url`.
- If the API contains multiple `basePath` variables, the Import API feature uses only the first one as the base path.

Ignore

If the OpenAPI file has a `basePath` value of `/a/b/c` and the `paths` property contains `/e` and `/f`, the following `POST` or `PUT` request:

```
POST /restapis?mode=import&basepath=ignore
```

```
PUT /restapis/api_id?basepath=ignore
```

results in the following resources in the API:

- /
- /e
- /f

The effect is to treat the basePath as if it was not present, and all of the declared API resources are served relative to the host. This can be used, for example, when you have a custom domain name with an API mapping that doesn't include a *Base Path* and a *Stage* value that refers to your production stage.

Note

API Gateway automatically creates a root resource for you, even if it isn't explicitly declared in your definition file.

When unspecified, basePath takes ignore by default.

Prepend

If the OpenAPI file has a basePath value of /a/b/c and the paths property contains /e and /f, the following POST or PUT request:

```
POST /restapis?mode=import&basepath=prepend
```

```
PUT /restapis/api_id?basepath=prepend
```

results in the following resources in the API:

- /
- /a
- /a/b
- /a/b/c
- /a/b/c/e
- /a/b/c/f

The effect is to treat the basePath as specifying additional resources (without methods) and to add them to the declared resource set. This can be used, for example, when different teams are responsible for different parts of an API and the basePath could reference the path location for each team's API part.

> **Note**
>
> API Gateway automatically creates intermediate resources for you, even if they aren't explicitly declared in your definition.

Split

If the OpenAPI file has a basePath value of /a/b/c and the paths property contains /e and /f, the following POST or PUT request:

```
POST /restapis?mode=import&basepath=split
```

```
PUT /restapis/api_id?basepath=split
```

results in the following resources in the API:

- /
- /b
- /b/c
- /b/c/e
- /b/c/f

The effect is to treat top-most path part, /a, as the beginning of each resource's path, and to create additional (no method) resources within the API itself. This could, for example, be used when a is a stage name that you want to expose as part of your API.

AWS variables for OpenAPI import

You can use the following AWS variables in OpenAPI definitions. API Gateway resolves the variables when the API is imported. To specify a variable, use ${*variable-name*}.

AWS variables

Variable name	Description	
`AWS::AccountId`	The AWS account ID that imports the API—for example, 123456789012.	
`AWS::Partition`	The AWS partition in which the API is imported. For standard AWS Regions, the partition is `aws`.	
`AWS::Region`	The AWS Region in which the API is imported—for example, `us-east-2` .	

AWS variables example

The following example uses AWS variables to specify an AWS Lambda function for an integration.

OpenAPI 3.0

```
openapi: "3.0.1"
info:
  title: "tasks-api"
  version: "v1.0"
paths:
  /:
    get:
      summary: List tasks
      description: Returns a list of tasks
      responses:
        200:
          description: "OK"
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/Task"
        500:
          description: "Internal Server Error"
          content: {}
      x-amazon-apigateway-integration:
        uri:
          arn:${AWS::Partition}:apigateway:${AWS::Region}:lambda:path/2015-03-31/
functions/arn:${AWS::Partition}:lambda:${AWS::Region}:
${AWS::AccountId}:function:LambdaFunctionName/invocations
        responses:
          default:
            statusCode: "200"
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        contentHandling: "CONVERT_TO_TEXT"
        type: "aws_proxy"
components:
  schemas:
    Task:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        description:
          type: string
```

Errors and warnings during import

Errors during import

During the import, errors can be generated for major issues like an invalid OpenAPI document. Errors are returned as exceptions (for example, `BadRequestException`) in an unsuccessful response. When an error occurs, the new API definition is discarded and no change is made to the existing API.

Warnings during import

During the import, warnings can be generated for minor issues like a missing model reference. If a warning occurs, the operation will continue if the `failonwarnings=false` query expression is appended to the request URL. Otherwise, the updates will be rolled back. By default, failonwarnings is set to false. In such cases, warnings are returned as a field in the resulting RestApi resource. Otherwise, warnings are returned as a message in the exception.

Export a REST API from API Gateway

Once you created and configured a REST API in API Gateway, using the API Gateway console or otherwise, you can export it to an OpenAPI file using the API Gateway Export API, which is part of the Amazon API Gateway Control Service. To use the API Gateway Export API, you need to sign your API requests. For more information about signing requests, see Signing AWS API requests in the *IAM User Guide*. You have options to include the API Gateway integration extensions, as well as the Postman extensions, in the exported OpenAPI definition file.

Note

When exporting the API using the AWS CLI, be sure to include the extensions parameter as shown in the following example, to ensure that the x-amazon-apigateway-request-validator extension is included:

```
aws apigateway get-export --parameters extensions='apigateway' --rest-api-id
 abcdefg123 --stage-name dev --export-type swagger latestswagger2.json
```

You cannot export an API if its payloads are not of the application/json type. If you try, you will get an error response stating that JSON body models are not found.

Request to export a REST API

With the Export API, you export an existing REST API by submitting a GET request, specifying the to-be-exported API as part of URL paths. The request URL is of the following format:

OpenAPI 3.0

```
https://<host>/restapis/<restapi_id>/stages/<stage_name>/exports/oas30
```

OpenAPI 2.0

```
https://<host>/restapis/<restapi_id>/stages/<stage_name>/exports/swagger
```

You can append the `extensions` query string to specify whether to include API Gateway extensions (with the `integration` value) or Postman extensions (with the `postman` value).

In addition, you can set the `Accept` header to `application/json` or `application/yaml` to receive the API definition output in JSON or YAML format, respectively.

For more information about submitting GET requests using the API Gateway Export API, see GetExport.

Note

If you define models in your API, they must be for the content type of "application/json" for API Gateway to export the model. Otherwise, API Gateway throws an exception with the "Only found non-JSON body models for ..." error message.
Models must contain properties or be defined as a particular JSONSchema type.

Download REST API OpenAPI definition in JSON

To export and download a REST API in OpenAPI definitions in JSON format:

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

Here, *<region>* could be, for example, `us-east-1`. For all the regions where API Gateway is available, see Regions and Endpoints

Download REST API OpenAPI definition in YAML

To export and download a REST API in OpenAPI definitions in YAML format:

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

Download REST API OpenAPI definition with Postman extensions in JSON

To export and download a REST API in OpenAPI definitions with Postman in JSON format:

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

Download REST API OpenAPI definition with API Gateway integration in YAML

To export and download a REST API in OpenAPI definitions with API Gateway integration in YAML format:

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30?extensions=integrations
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

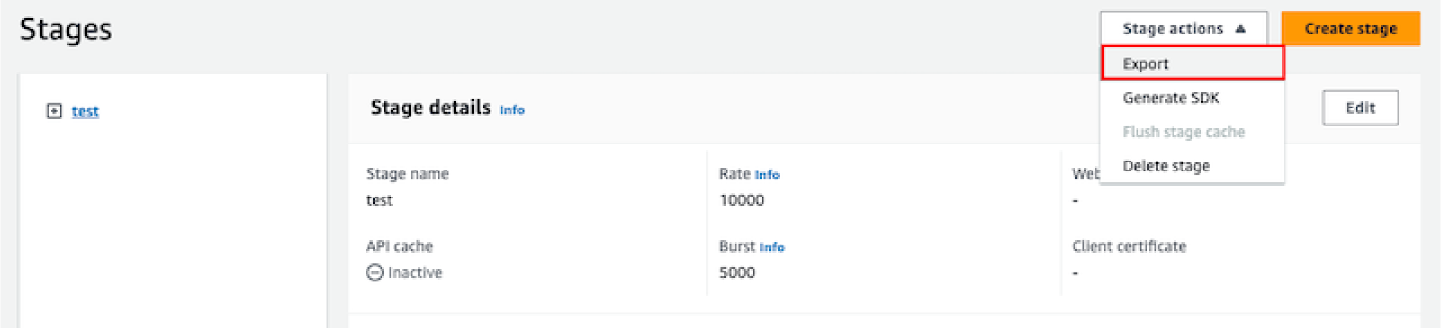
OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?
extensions=integrations
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

Export REST API using the API Gateway console

After deploying your REST API to a stage, you can proceed to export the API in the stage to an OpenAPI file using the API Gateway console.

In the **Stages** pane in the API Gateway console, choose **Stage actions**, **Export**.



Specify an **API specification type**, **Format**, and **Extensions** to download your API's OpenAPI definition.

Publishing REST APIs for customers to invoke

Simply creating and developing an API Gateway API doesn't automatically make it callable by your users. To make it callable, you must deploy your API to a stage. In addition, you might want to customize the URL that your users will use to access your API. You can give it a domain that is consistent with your brand or is more memorable than the default URL for your API.

In this section, you can learn how to deploy your API and customize the URL that you provide to users to access it.

Note

To augment the security of your API Gateway APIs, the `execute-api.{region}.amazonaws.com` domain is registered in the Public Suffix List (PSL). For further security, we recommend that you use cookies with a `__Host-` prefix if you ever need to set sensitive cookies in the default domain name for your API Gateway APIs. This practice will help to defend your domain against cross-site request forgery attempts (CSRF). For more information see the Set-Cookie page in the Mozilla Developer Network.

Topics

Deploying a REST API in Amazon API Gateway

After creating your API, you must deploy it to make it callable by your users.

To deploy an API, you create an API deployment and associate it with a stage. A stage is a logical reference to a lifecycle state of your API (for example, `dev`, `prod`, `beta`, `v2`). API stages are identified by the API ID and stage name. They're included in the URL that you use to invoke the API. Each stage is a named reference to a deployment of the API and is made available for client applications to call.

> ⚠ **Important**
>
> Every time you update an API, you must redeploy the API to an existing stage or to a new stage. Updating an API includes modifying routes, methods, integrations, authorizers, and anything else other than stage settings.

As your API evolves, you can continue to deploy it to different stages as different versions of the API. You can also deploy your API updates as a canary release deployment. This enables your API clients to access, on the same stage, the production version through the production release, and the updated version through the canary release.

To call a deployed API, the client submits a request against an API's URL. The URL is determined by an API's protocol (HTTP(S) or (WSS)), hostname, stage name, and (for REST APIs) resource path. The hostname and the stage name determine the API's base URL.

Using the API's default domain name, the base URL of a REST API (for example) in a given stage (*`{stageName}`*) is in the following format:

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stageName}
```

To make the API's default base URL more user-friendly, you can create a custom domain name (for example, `api.example.com`) to replace the default hostname of the API. To support multiple APIs under the custom domain name, you must map an API stage to a base path.

With a custom domain name of *`{api.example.com}`* and the API stage mapped to a base path of (*`{basePath}`*) under the custom domain name, the base URL of a REST API becomes the following:

```
https://{api.example.com}/{basePath}
```

For each stage, you can optimize API performance by adjusting the default account-level request throttling limits and enabling API caching. You can also enable logging for API calls to CloudTrail or CloudWatch, and can select a client certificate for the backend to authenticate the API requests. In addition, you can override stage-level settings for individual methods and define stage variables to pass stage-specific environment contexts to the API integration at runtime.

Stages enable robust version control of your API. For example, you can deploy an API to a `test` stage and a `prod` stage, and use the `test` stage as a test build and use the `prod` stage as a stable build. After the updates pass the test, you can promote the `test` stage to the `prod` stage. The promotion can be done by redeploying the API to the `prod` stage or updating a stage variable value from the stage name of `test` to that of `prod`.

In this section, we discuss how to deploy an API by using the API Gateway console or calling the API Gateway REST API. To use other tools, see the documentation of the AWS CLI or an AWS SDK.

Topics

- Deploy a REST API in API Gateway
- Setting up a stage for a REST API
- Set up an API Gateway canary release deployment
- Updates to a REST API that require redeployment

Deploy a REST API in API Gateway

In API Gateway, a REST API deployment is represented by a Deployment resource. It's similar to an executable of an API that is represented by a RestApi resource.

For the client to call your API, you must create a deployment and associate a stage with it. A stage is represented by a Stage resource. It represents a snapshot of the API, including methods, integrations, models, mapping templates, and Lambda authorizers (formerly known as custom authorizers). When you update the API, you can redeploy the API by associating a new stage with the existing deployment. We discuss creating a stage in the section called “Set up a stage”.

Topics

- Create a deployment using the AWS CLI

- Deploying a REST API from the API Gateway console

Create a deployment using the AWS CLI

When you create a deployment, you instantiate the Deployment resource. You can use the API Gateway console, the AWS CLI, an AWS SDK, or the API Gateway REST API to create a deployment.

To use the CLI to create a deployment, use the create-deployment command:

```
aws apigateway create-deployment --rest-api-id <rest-api-id> --region <region>
```

The API is not callable until you associate this deployment with a stage. With an existing stage, you can do this by updating the stage's `deploymentId` property with the newly created deployment ID (`<deployment-id>`).

```
aws apigateway update-stage --region <region> \
   --rest-api-id <rest-api-id> \
   --stage-name <stage-name> \
   --patch-operations op='replace',path='/deploymentId',value='<deployment-id>'
```

When deploying an API the first time, you can combine the stage creation and deployment creation at the same time:

```
aws apigateway create-deployment --region <region> \
   --rest-api-id <rest-api-id> \
   --stage-name <stage-name>
```

This is what is done behind the scenes in the API Gateway console when you deploy an API the first time, or when you redeploy the API to a new stage.

Deploying a REST API from the API Gateway console

You must have created a REST API before deploying it for the first time. For more information, see Creating a REST API in Amazon API Gateway.

Topics

- Update the stage configuration of a REST API deployment
- Set stage variables for a REST API deployment
- Associate a stage with a different REST API deployment

Deploy a REST API to a stage

The API Gateway console lets you deploy an API by creating a deployment and associating it with a new or existing stage.

> **Note**
>
> To associate a stage in API Gateway with a different deployment, see Associate a stage with a different REST API deployment instead.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the **APIs** navigation pane, choose the API you want to deploy.
3. In the **Resources** pane, choose **Deploy API**.
4. For **Stage**, select from the following:

   a. To create a new stage, select **New stage**, and then enter a name in **Stage name**. Optionally, you can provide a description for the deployment in **Deployment description**.

   b. To choose an existing stage, select the stage name from the dropdown menu. You might want to provide a description of the new deployment in **Deployment description**.

   c. To create a deployment that is not associated with a stage, select **No stage**. Later, you can associate this deployment with a stage.
5. Choose **Deploy**.

Redeploy a REST API to a stage

To redeploy an API, perform the same steps as in the section called "Deploy a REST API to a stage". You can reuse the same stage as many times as desired.

Update the stage configuration of a REST API deployment

After an API is deployed, you can modify the stage settings to enable or disable the API cache, logging, or request throttling. You can also choose a client certificate for the backend to authenticate API Gateway and set stage variables to pass deployment context to the API integration at runtime. For more information, see Update stage settings.

⚠ **Important**

After modifying stage settings, you must redeploy the API for the changes to take effect.

ⓘ **Note**

If the updated settings, such as enabling logging, requires a new IAM role, you can add the required IAM role without redeploying the API. However, it can take a few minutes before the new IAM role takes effect. Before that happens, traces of your API calls are not logged even if you have enabled the logging option.

Set stage variables for a REST API deployment

For a deployment, you can set or modify stage variables to pass deployment-specific data to the API integration at runtime. You can do this on the **Stage Variables** tab in the **Stage Editor**. For more information, see instructions in Setting up stage variables for a REST API deployment.

Associate a stage with a different REST API deployment

Because a deployment represents an API snapshot and a stage defines a path into a snapshot, you can choose different deployment-stage combinations to control how users call into different versions of the API. This is useful, for example, when you want to roll back API state to a previous deployment or to merge a 'private branch' of the API into the public one.

The following procedure shows how to do this using the **Stage Editor** in the API Gateway console. It is assumed that you must have deployed an API more than once.

1. If you're not already on the **Stages** pane, in the main navigation pane, choose **Stages**.
2. Select the stage you want to update.
3. On the **Deployment history** tab, select the deployment you want the stage to use.
4. Choose **Change active deployment**.
5. Confirm you want to change the active deployment and choose **Change active deployment** in the **Make active deployment** dialog box.

Setting up a stage for a REST API

A stage is a named reference to a deployment, which is a snapshot of the API. You use a Stage to manage and optimize a particular deployment. For example, you can configure stage settings to enable caching, customize request throttling, configure logging, define stage variables, or attach a canary release for testing.

Topics

- Setting up a stage using the API Gateway console
- Setting up tags for an API stage in API Gateway
- Setting up stage variables for a REST API deployment

Setting up a stage using the API Gateway console

Topics

- Create a new stage
- Update stage settings
- Override stage-level settings
- Delete a stage

Create a new stage

After the initial deployment, you can add more stages and associate them with existing deployments. You can use the API Gateway console to create a new stage, or you can choose an existing stage while deploying an API. In general, you can add a new stage to an API deployment before redeploying the API. To create a new stage using the API Gateway console, follow these steps:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **Stages** under an API.
4. From the **Stages** navigation pane, choose **Create stage**.
5. For **Stage name**, enter a name, for example, **`prod`**.

> **Note**
>
> Stage names can only contain alphanumeric characters, hyphens, and underscores. Maximum length is 128 characters.

6. (Optional). For **Description**, enter a stage description.
7. For **Deployment**, select the date and time of the existing API deployment you want to associate with this stage.
8. Under **Additional settings**, you can specify additional settings for your stage.
9. Choose **Create stage**.

Update stage settings

After a successful deployment of an API, the stage is populated with default settings. You can use the console or the API Gateway REST API to change the stage settings, including API caching and logging. The following steps show you how to do so using the **Stage editor** of the API Gateway console.

Update stage settings using the API Gateway console

These steps assume that you've already deployed the API to a stage.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **Stages** under an API.
4. In the **Stages** pane, choose the name of the stage.
5. In the **Stage details** section, choose **Edit**.
6. (Optional) For **Stage description**, edit the description.
7. For **Additional settings**, you modify the following settings:

   Cache settings

   To enable API caching for the stage, turn on **Provision API cache**. Then configure the **Default method-level caching**, **Cache capacity**, **Encrypt cache data**, **Cache time-to-live (TTL)**, as well as any requirements for per-key cache invalidation.

Caching is not active until you turn on the default method-level caching or turn on the method-level cache for a specific method.

For more information about cache settings, see Enabling API caching to enhance responsiveness.

> **Note**
>
> If you enable API caching for an API stage, your AWS account might be charged for API caching. Caching isn't eligible for the AWS Free Tier.

Throttling settings

To set stage-level throttling targets for all of the methods associated with this API, turn on **Throttling**.

For **Rate**, enter a target rate. This is the rate, in requests per second, that tokens are added to the token bucket. The stage-level rate must not be more than the account-level rate as specified in API Gateway quotas for configuring and running a REST API.

For **Burst**, enter a target burst rate. The burst rate, is the capacity of the token bucket. This allows more requests through for a period of time than the target rate. This stage-level burst rate must not be more than the account-level burst rate as specified in API Gateway quotas for configuring and running a REST API.

> **Note**
>
> Throttling rates are not hard limits, and are applied on a best-effort basis. In some cases, clients can exceed the targets that you set. Don’t rely on throttling to control costs or block access to an API. Consider using AWS Budgets to monitor costs and AWS WAF to manage API requests.

Firewall and certificate settings

To associate an AWS WAF web ACL with the stage, select a web ACL from the **Web ACL** dropdown list. If desired, choose **Block API Request if WebACL cannot be evaluated (Fail-Close)**.

To select a client certificate for your stage, select a certificate from the **Client certificate** dropdown menu.

8. Choose **Save**.
9. To enable Amazon CloudWatch Logs for all of the methods associated with this stage of this API Gateway API, in the **Logs and tracing** section, choose **Edit**.

Note

To enable CloudWatch Logs, you must also specify the ARN of an IAM role that enables API Gateway to write information to CloudWatch Logs on behalf of your user. To do so, choose **Settings** from the **APIs** main navigation pane. Then, for **CloudWatch log role**, enter the ARN of an IAM role.
For common application scenarios, the IAM role could attach the managed policy of `AmazonAPIGatewayPushToCloudWatchLogs`, which contains the following access policy statement:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:GetLogEvents",
                "logs:FilterLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

The IAM role must also contain the following trust relationship statement:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

For more information about CloudWatch, see the *Amazon CloudWatch User Guide*.

10. Select a logging level from the **CloudWatch Logs** dropdown menu. The logging levels are the following:

    - **Off** – Logging is not turned on for this stage.
    - **Errors only** – Logging is enabled for errors only.
    - **Errors and info logs** – Logging is enabled for all events.
    - **Full request and response logs** – Detailed logging is enabled for all events. This can be useful to troubleshoot APIs, but can result in logging sensitive data.

    Note

    We recommend that you don't use **Full request and response logs** for production APIs.

11. Select **Detailed metrics** to have API Gateway report to CloudWatch the API metrics of `API calls`, `Latency`, `Integration latency`, `400 errors`, and `500 errors`. For more information about CloudWatch, see the Basic monitoring and detailed monitoring in the Amazon CloudWatch User Guide.

> ⚠ **Important**
>
> Your account is charged for accessing method-level CloudWatch metrics, but not the API-level or stage-level metrics.

12. To enable access logging to a destination, turn on **Custom access logging**.
13. For **Access log destination ARN**, enter the ARN of a log group or a Firehose stream.

    The ARN format for Firehose is `arn:aws:firehose:{region}:{account-id}:deliverystream/amazon-apigateway-{your-stream-name}`. The name of your Firehose stream must be `amazon-apigateway-{your-stream-name}`.
14. In **Log format**, enter a log format. To learn more about example log formats, see the section called "CloudWatch log formats for API Gateway".
15. To enable AWS X-Ray tracing for the API stage, select **X-Ray tracing**. For more information, see Tracing user requests to REST APIs using X-Ray.
16. Choose **Save changes**. Redeploy your API for the new settings to take effect.

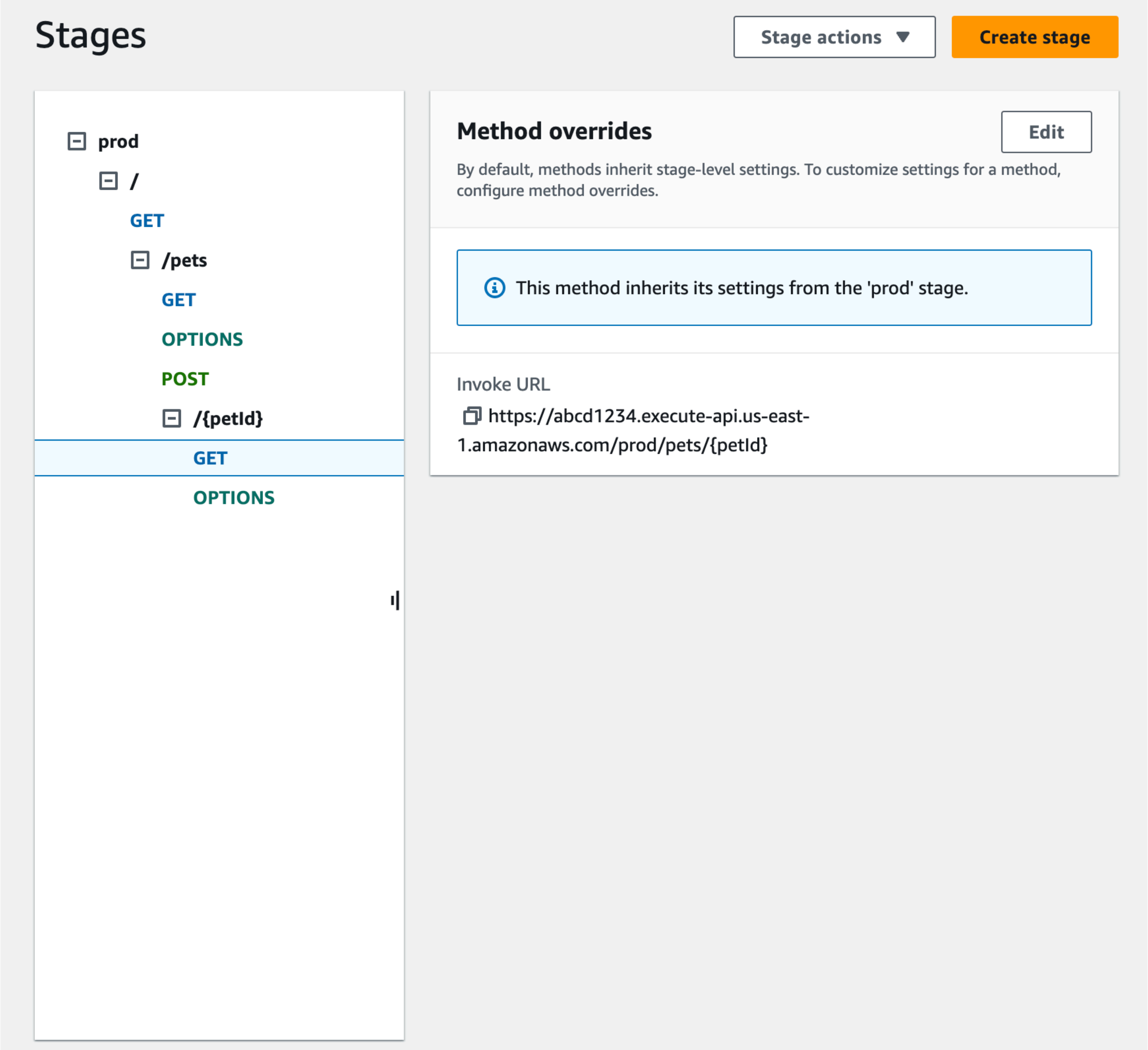
Override stage-level settings

You can override the following enabled stage-level settings. Some of these options might result in additional charges to your AWS account.

Override stage-level settings using the API Gateway console

To override stage-level settings using the API Gateway console

1. To configure method overrides, expand the stage under the secondary navigation pane, and then choose a method.



Method overrides

By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.

This method inherits its settings from the 'prod' stage.

Invoke URL

https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/pets/{petId}

2. For **Method overrides**, choose **Edit**.
3. To turn on method-level CloudWatch settings, for **CloudWatch Logs**, select a logging level.
4. To turn on method-level detailed metrics, select **Detailed metrics**. Your account is charged for accessing method-level CloudWatch metrics, but not the API-level or stage-level metrics.
5. To turn on method-level throttling, select **Throttling**. Enter the appropriate method-level options. To learn more about throttling, see the section called “Throttling”.
6. To configure the method-level cache, select **Enable method cache**. If you change the default method-level caching setting in the **Stage details**, it doesn't affect this setting.
7. Choose **Save**.

Delete a stage

When you no longer need a stage, you can delete it to avoid paying for unused resources. The following steps show you how to use the API Gateway console to delete a stage.

> ⚠ **Warning**
>
> Deleting a stage might cause part or all of the corresponding API to be unusable by API callers. Deleting a stage cannot be undone, but you can recreate the stage and associate it with the same deployment.

Delete a stage using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **Stages**.
4. In the **Stages** pane, choose the stage you want to delete, and then choose **Stage actions**, **Delete stage**.
5. When you're prompted, enter **`confirm`**, and then choose **Delete**.

Setting up tags for an API stage in API Gateway

In API Gateway, you can add a tag to an API stage, remove the tag from the stage, or view the tag. To do this, you can use the API Gateway console, the AWS CLI/SDK, or the API Gateway REST API.

A stage can also inherit tags from its parent REST API. For more information, see the section called “Tag inheritance in the Amazon API Gateway V1 API”.

For more information about tagging API Gateway resources, see *Tagging*.

Topics

Set up tags for an API stage using the API Gateway console

The following procedure describes how to set up tags for an API stage.

To set up tags for an API stage by using the API Gateway console

1. Sign in to the API Gateway console.
2. Choose an existing API, or create a new API that includes resources, methods, and the corresponding integrations.
3. Choose a stage or deploy the API to a new stage.
4. In the main navigation pane, choose **Stages**.
5. Choose the **Tags** tab. You might need to choose the right arrow button to show the tab.
6. Choose **Manage tags**.
7. In the **Tag Editor**, choose **Add tag**. Enter a tag key (for example, `Department`) in the **Key** field, and enter a tag value (for example, `Sales`) in the **Value** field. Choose **Save** to save the tag.
8. If needed, repeat step 5 to add more tags to the API stage. The maximum number of tags per stage is 50.
9. To remove an existing tag from the stage, choose **Remove**.
10. If the API has been deployed previously in the API Gateway console, you need to redeploy it for the changes to take effect.

Set up tags for an API stage using the AWS CLI

You can set up tags for an API stage using the AWS CLI using the create-stage command or the tag-resource command. You can delete one or more tags from an API stage using the untag-resource command.

The following example adds a tag when creating a `test` stage:

```
aws apigateway create-stage --rest-api-id abc1234 --stage-name test --description
 'Testing stage' --deployment-id efg456 --tag Department=Sales
```

The following example adds a tag to a `prod` stage:

```
aws apigateway tag-resource --resource-arn arn:aws:apigateway:us-east-2::/
restapis/abc123/stages/prod --tags Department=Sales
```

The following example removes the Department=Sales tag from the test stage:

```
aws apigateway untag-resource --resource-arn arn:aws:apigateway:us-east-2::/
restapis/abc123/stages/test --tag-keys Department
```

Set up tags for an API stage using the API Gateway REST API

You can set up tags for an API stage using the API Gateway REST API by doing one of the following:

- Call tags:tag to tag an API stage.
- Call tags:untag to delete one or more tags from an API stage.
- Call stage:create to add one or more tags to an API stage that you're creating.

You can also call tags:get to describe tags in an API stage.

Tag an API stage

After you deploy an API (m5zr3vnks7) to a stage (test), tag the stage by calling tags:tag. The required stage Amazon Resource Name (ARN) (arn:aws:apigateway:us-east-1::/restapis/m5zr3vnks7/stages/test) must be URL encoded (arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages%2Ftest).

```
PUT /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftest

{
  "tags" : {
    "Department" : "Sales"
  }
}
```

You can also use the previous request to update an existing tag to a new value.

You can add tags to a stage when calling stage:create to create the stage:

```
POST /restapis/<restapi_id>/stages

{
  "stageName" : "test",
  "deploymentId" : "adr134",
  "description" : "test deployment",
  "cacheClusterEnabled" : "true",
  "cacheClusterSize" : "500",
  "variables" : {
    "sv1" : "val1"
  },
  "documentationVersion" : "test",

  "tags" : {
    "Department" : "Sales",
    "Division" : "Retail"
  }
}
```

Untag an API stage

To remove the `Department` tag from the stage, call [`tags:untag`]:

```
DELETE /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftest?tagKeys=Department
Host: apigateway.us-east-1.amazonaws.com
Authorization: ...
```

To remove more than one tag, use a comma-separated list of tag keys in the query expression—for example, `?tagKeys=Department,Division,…`.

Describe tags for an API stage

To describe existing tags on a given stage, call [`tags:get`]:

```
GET /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftags
Host: apigateway.us-east-1.amazonaws.com
Authorization: ...
```

The successful response is similar to the following:

```
200 OK

{
    "_links": {
        "curies": {
            "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/
restapi-tags-{rel}.html",
            "name": "tags",
            "templated": true
        },
        "tags:tag": {
            "href": "/tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis
%2Fm5zr3vnks7%2Fstages%2Ftags"
        },
        "tags:untag": {
            "href": "/tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis
%2Fm5zr3vnks7%2Fstages%2Ftags{?tagKeys}",
            "templated": true
        }
    },
    "tags": {
        "Department": "Sales"
    }
}
```

Setting up stage variables for a REST API deployment

Stage variables are name-value pairs that you can define as configuration attributes associated with a deployment stage of a REST API. They act like environment variables and can be used in your API setup and mapping templates.

For example, you can define a stage variable in a stage configuration, and then set its value as the URL string of an HTTP integration for a method in your REST API. Later, you can reference the URL string by using the associated stage variable name from the API setup. By doing this, you can use the same API setup with a different endpoint at each stage by resetting the stage variable value to the corresponding URLs.

You can also access stage variables in the mapping templates, or pass configuration parameters to your AWS Lambda or HTTP backend.

For more information about mapping templates, see API Gateway mapping template and access logging variable reference.

Note

Stage variables are not intended to be used for sensitive data, such as credentials. To pass sensitive data to integrations, use an AWS Lambda authorizer. You can pass sensitive data to integrations in the output of the Lambda authorizer. To learn more, see the section called “Output from an Amazon API Gateway Lambda authorizer”.

Use cases

With deployment stages in API Gateway, you can manage multiple release stages for each API, such as alpha, beta, and production. Using stage variables you can configure an API deployment stage to interact with different backend endpoints.

For example, your API can pass a GET request as an HTTP proxy to the backend web host (for example, `http://example.com`). In this case, the backend web host is configured in a stage variable so that when developers call your production endpoint, API Gateway calls example.com. When you call your beta endpoint, API Gateway uses the value configured in the stage variable for the beta stage, and calls a different web host (for example, `beta.example.com`). Similarly, stage variables can be used to specify a different AWS Lambda function name for each stage in your API.

You can also use stage variables to pass configuration parameters to a Lambda function through your mapping templates. For example, you might want to reuse the same Lambda function for multiple stages in your API, but the function should read data from a different Amazon DynamoDB table depending on which stage is being called. In the mapping templates that generate the request for the Lambda function, you can use stage variables to pass the table name to Lambda.

Examples

To use a stage variable to customize the HTTP integration endpoint, you must first configure a stage variable of a specified name (for example, **`url`**), and then assign it a value, (for example, **`example.com`**). Next, from your method configuration, set up an HTTP proxy integration. Instead of entering the endpoint's URL, you can tell API Gateway to use the stage variable value, **`http://${stageVariables.url}`**. This value tells API Gateway to substitute your stage variable `${}` at runtime, depending on which stage your API is running.

You can reference stage variables in a similar way to specify a Lambda function name, an AWS Service Proxy path, or an AWS role ARN in the credentials field.

When specifying a Lambda function name as a stage variable value, you must configure the permissions on the Lambda function manually. When you specify a Lambda function in the API Gateway console, a AWS CLI command will pop-up to configure the proper permissions. You can also use the AWS Command Line Interface (AWS CLI) to do this.

```
aws lambda add-permission --function-name "arn:aws:lambda:us-east-2:123456789012:function:my-function" --source-arn "arn:aws:execute-api:us-east-2:123456789012:api_id/*/HTTP_METHOD/resource" --principal apigateway.amazonaws.com
 --statement-id apigateway-access --action lambda:InvokeFunction
```

Setting stage variables using the Amazon API Gateway console

In this tutorial, you learn how to set stage variables for two deployment stages of a sample API by using the Amazon API Gateway console. Before you begin, make sure the following prerequisites are met:

- You must have an API available in API Gateway. Follow the instructions in Creating a REST API in Amazon API Gateway.
- You must have deployed the API at least once. Follow the instructions in Deploying a REST API in Amazon API Gateway.
- You must have created the first stage for a deployed API. Follow the instructions in Create a new stage.

To declare stage variables using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Create an API, and then create a GET method on the API's root resource. Set the integration type to **HTTP** and set the **Endpoint URL** to **`http://${stageVariables.url}`**.
3. Deploy the API to a new stage named **`beta`**.
4. In the main navigation pane, choose **Stages**, and then select the **beta** stage.
5. On the **Stage variables** tab, choose **Edit**.
6. Choose **Add stage variable**.
7. For **Name**, enter **`url`**. For **value**, enter **`httpbin.org/get`**.
8. Choose **Add stage variable**, and then do the following:

   For **Name**, enter **`stageName`**. For **value**, enter **`beta`**.
9. Choose **Add stage variable**, and then do the following:

   For **Name**, enter **`function`**. For **value**, enter **`HelloWorld`**.

Note

When setting a Lambda function as the value of a stage variable, use the function's local name, possibly including its alias or version specification, as in **`HelloWorld`**, **`HelloWorld:1`** or **`HelloWorld:alpha`**. Do not use the function's ARN (for example, **`arn:aws:lambda:us-east-1:123456789012:function:HelloWorld`**). The API Gateway console assumes the stage variable value for a Lambda function as the unqualified function name and expands the given stage variable into an ARN.

10. Choose **Save**.
11. Now create a second stage. From the **Stages** navigation pane, choose **Create stage**. For **Stage name**, enter **`prod`**. Select a recent deployment from **Deployment**, and then choose **Create stage**.
12. As with the **beta** stage, set the same three stage variables (**url**, **stageName**, and **function**) to different values (**`petstore-demo-endpoint.execute-api.com/petstore/pets`**, **`prod`**, and **`HelloEveryone`**), respectively.

    To learn how to use stage variables, see the section called "Use stage variables ".

Using Amazon API Gateway stage variables

You can use API Gateway stage variables to access the HTTP and Lambda backends for different API deployment stages. You can also use stage variables to pass stage-specific configuration metadata into an HTTP backend as a query parameter and into a Lambda function as a payload that is generated in an input mapping template.

Prerequisites

You must create two stages with a **url** stage variable set to two different HTTP endpoints: a **function** stage variable assigned to two different Lambda functions, and a **stageName** stage variable containing stage-specific metadata.

Access an HTTP endpoint through an API with a stage variable

1. In the **Stages** navigation pane, choose **beta**. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in a web browser. This starts the **beta** stage GET request on the root resource of the API.

> **Note**
>
> The **Invoke URL** link points to the root resource of the API in its **beta** stage. Entering the URL in a web browser calls the **beta** stage `GET` method on the root resource. If methods are defined on child resources and not on the root resource itself, entering the URL in a web browser returns a `{"message":"Missing Authentication Token"}` error response. In this case, you must append the name of a specific child resource to the **Invoke URL** link.

2. The response you get from the **beta** stage `GET` request is shown next. You can also verify the result by using a browser to navigate to **http://httpbin.org/get**. This value was assigned to the `url` variable in the **beta** stage. The two responses are identical.
3. In the **Stages** navigation pane, choose the **prod** stage. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in a web browser. This starts the **prod** stage `GET` request on the root resource of the API.
4. The response you get from the **prod** stage `GET` request is shown next. You can verify the result by using a browser to navigate to **http://petstore-demo-endpoint.execute-api.com/petstore/pets**. This value was assigned to the `url` variable in the **prod** stage. The two responses are identical.

Pass stage-specific metadata to an HTTP backend through a stage variable in a query parameter expression

This procedure describes how to use a stage variable value in a query parameter expression to pass stage-specific metadata into an HTTP backend. We will use the `stageName` stage variable declared in Setting stage variables using the Amazon API Gateway console.

1. In the **Resource** navigation pane, choose the **GET** method.

   To add a query string parameter to the method's URL, choose the **Method request** tab, and then in the **Method request settings** section, choose **Edit**.
2. Choose **URL query string parameters** and do the following:

   a. Choose **Add query string**.

   b. For **Name**, enter **`stageName`**.

   c. Keep **Required** and **Caching** turned off.

3. Choose **Save**.

4. Choose the **Integration request** tab, and then in the **Integration request settings** section, choose **Edit**.

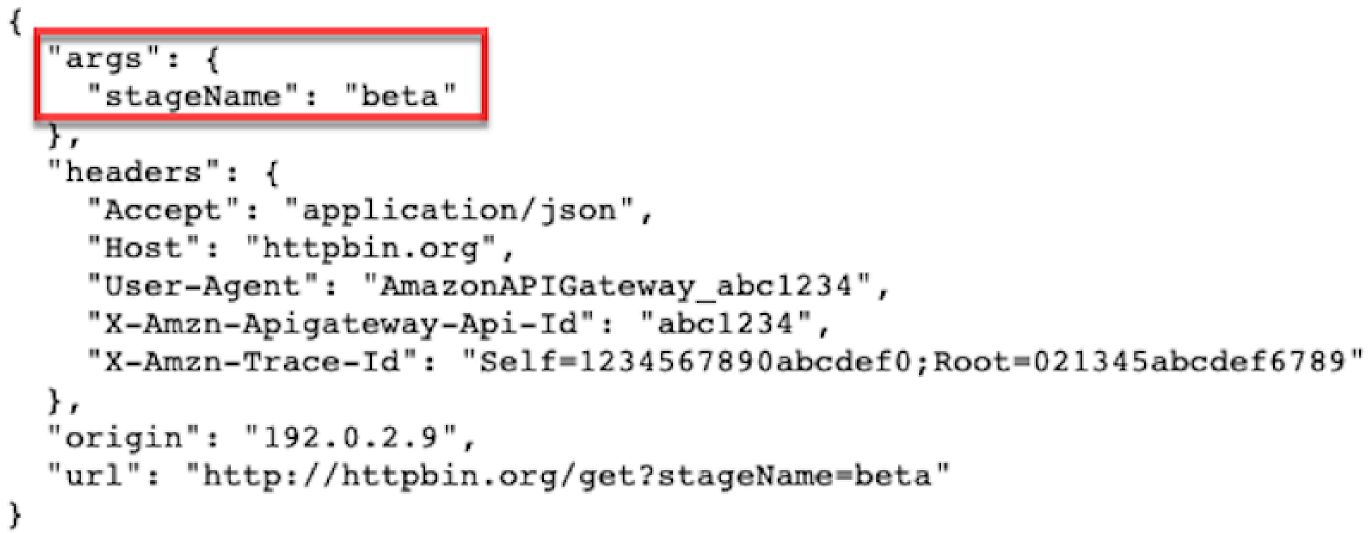
5. For **Endpoint URL**, append **`?stageName=${stageVariables.stageName}`** to the previously defined URL value, so the entire **Endpoint URL** is **`http://${stageVariables.url}?stageName=${stageVariables.stageName}`**.

6. Choose **Deploy API** and select the **beta** stage.

7. In the main navigation pane, choose **Stages**. In the **Stages** navigation pane, choose **beta**. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in a web browser.

> **Note**
>
> We use the beta stage here because the HTTP endpoint (as specified by the `url` variable "http://httpbin.org/get") accepts query parameter expressions and returns them as the `args` object in its response.

8. You get the following response. Notice that `beta`, assigned to the `stageName` stage variable, is passed in the backend as the `stageName` argument.



```
{
  "args": {
    "stageName": "beta"
  },
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "AmazonAPIGateway_abc1234",
    "X-Amzn-Apigateway-Api-Id": "abc1234",
    "X-Amzn-Trace-Id": "Self=1234567890abcdef0;Root=021345abcdef6789"
  },
  "origin": "192.0.2.9",
  "url": "http://httpbin.org/get?stageName=beta"
}
```

Call a Lambda function through an API with a stage variable

This procedure describes how to use a stage variable to call a Lambda function as a backend of your API. We will use the `function` stage variable declared earlier. For more information, see Setting stage variables using the Amazon API Gateway console.

1. Create a Lambda function named **`HelloWorld`** using the default Node.js runtime. The code must contain the following:

```
export const handler = function(event, context, callback) {
    if (event.stageName)
        callback(null, 'Hello, World! I\'m calling from the ' + event.stageName + '
 stage.');
    else
        callback(null, 'Hello, World! I\'m not sure where I\'m calling from...');
};
```

   For more information on how to create a Lambda function, see Getting started with the REST API console.

2. In the **Resources** pane, select **Create resource**, and then do the following:

   a. For **Resource path**, select **/**.

   b. For **Resource name**, enter **`lambdav1`**.

   c. Choose **Create resource**.

3. Choose the **/lambdav1** resource, and then choose **Create method**.

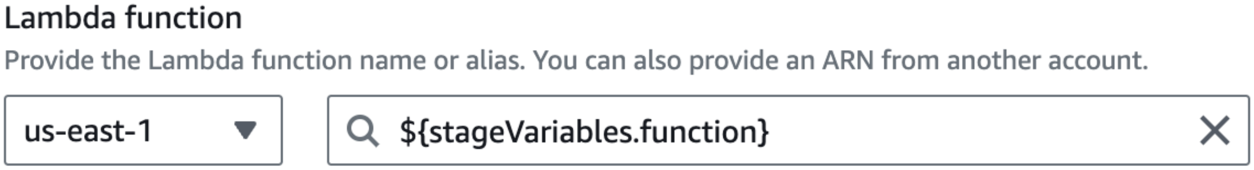
   Then, do the following:

   a. For **Method type**, select **GET**.

   b. For **Integration type**, select **Lambda function**.

   c. Keep **Lambda proxy integration** turned off.

   d. For **Lambda function**, enter `${stageVariables.function}`.



Lambda function

Provide the Lambda function name or alias. You can also provide an ARN from another account.

us-east-1

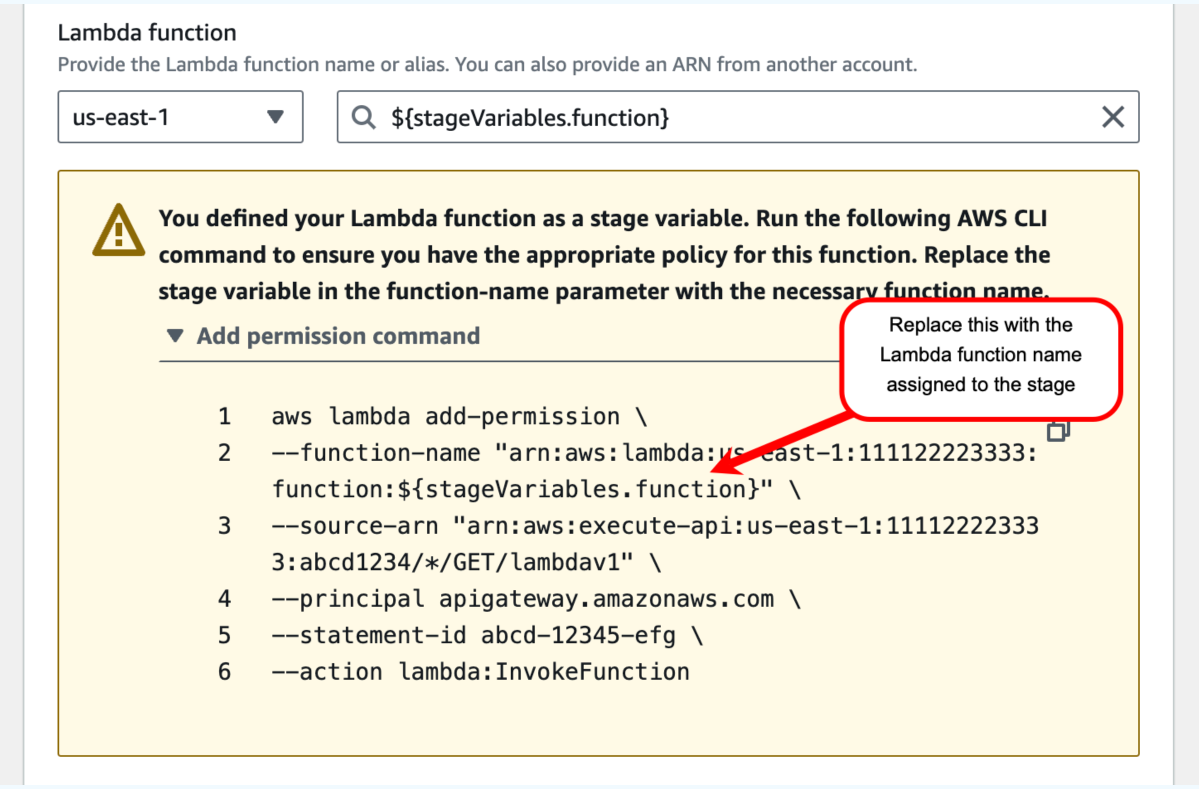
${stageVariables.function}

Tip

When prompted with the **Add permission command**, copy the AWS CLI command. Run the command on each Lambda function that will be assigned to the `function` stage variable. For example, if the `$stageVariables.function` value is `HelloWorld`, run the following AWS CLI command:

```
aws lambda add-permission --function-name arn:aws:lambda:us-
east-1:account-id:function:HelloWorld --source-arn arn:aws:execute-
api:us-east-1:account-id:api-id/*/GET/lambdav1 --principal
 apigateway.amazonaws.com --statement-id statement-id-guid --action
 lambda:InvokeFunction
```

Failing to do so results in a `500 Internal Server Error` response when invoking the method. Replace `${stageVariables.function}` with the Lambda function name that is assigned to the stage variable.



Lambda function

Provide the Lambda function name or alias. You can also provide an ARN from another account.

You defined your Lambda function as a stage variable. Run the following AWS CLI command to ensure you have the appropriate policy for this function. Replace the stage variable in the function-name parameter with the necessary function name.

▼ **Add permission command**

```
aws lambda add-permission \
--function-name "arn:aws:lambda:us-east-1:111122223333:
function:${stageVariables.function}" \
--source-arn "arn:aws:execute-api:us-east-1:11112222333
3:abcd1234/*/GET/lambdav1" \
--principal apigateway.amazonaws.com \
--statement-id abcd-12345-efg \
--action lambda:InvokeFunction
```

Replace this with the Lambda function name assigned to the stage

e. Choose **Create method**.

4. Deploy the API to both the **`prod`** and **`beta`** stages.

5. In the main navigation pane, choose **Stages**. In the **Stages** navigation pane, choose **beta**. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in a web browser. Append **`/lambdav1`** to the URL before you press enter.

   You get the following response.

   ```
   "Hello, World! I'm not sure where I'm calling from..."
   ```

Pass stage-specific metadata to a Lambda function through a stage variable

This procedure describes how to use a stage variable to pass stage-specific configuration metadata into a Lambda function. You create a `POST` method and an input mapping template to generate payload using the `stageName` stage variable you declared earlier.

1. Choose the **/lambdav1** resource, and then choose **Create method**.

   Then, do the following:

   a. For **Method type**, select **POST**.
   b. For **Integration type**, select **Lambda function**.
   c. Keep **Lambda proxy integration** turned off.
   d. For **Lambda function**, enter `${stageVariables.function}`.
   e. When prompted with the **Add permission command**, copy the AWS CLI command. Run the command on each Lambda function that will be assigned to the `function` stage variable.
   f. Choose **Create method**.
2. Choose the **Integration request** tab, and then in the **Integration request settings** section, choose **Edit**.
3. Choose **Mapping templates**, and then choose **Add mapping template**.
4. For **Content type**, enter **`application/json`**.
5. For **Template body**, enter the following template:

   ```
   #set($inputRoot = $input.path('$'))
   {
       "stageName" : "$stageVariables.stageName"
   }
   ```

> **Note**
>
> In a mapping template, a stage variable must be referenced within quotes (as in `"$stageVariables.stageName"` or `"${stageVariables.stageName}"`). In other places, it must be referenced without quotes (as in `${stageVariables.function}`).

6. Choose **Save**.
7. Deploy the API to both the **`beta`** and **`prod`** stages.
8. To use a REST API client to pass stage-specific metadata, do the following:

   a. In the **Stages** navigation pane, choose **beta**. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in the input field of a REST API client. Append **`/lambdav1`** before you submit your request.

      You get the following response.

      ```
      "Hello, World! I'm calling from the beta stage."
      ```

   b. In the **Stages** navigation pane, choose **prod**. Under **Stage details**, choose the copy icon to copy your API's invoke URL, and then enter your API's invoke URL in the input field of a REST API client. Append **`/lambdav1`** before you submit your request.

      You get the following response.

      ```
      "Hello, World! I'm calling from the prod stage."
      ```

9. To use the **Test** feature to pass stage-specific metadata, do the following:

   a. In the **Resources** navigation pane, choose the **Test** tab. You might need to choose the right arrow button to show the tab.

   b. For **function**, enter **`HelloWorld`**.

   c. For **stageName**, enter **`beta`**.

   d. Choose **Test**. You do not need to add a body to your `POST` request.

      You get the following response.

      ```
      "Hello, World! I'm calling from the beta stage."
      ```

e. You can repeat the previous steps to test the **Prod** stage. For **stageName**, enter **`Prod`**.

    You get the following response.

    ```
    "Hello, World! I'm calling from the prod stage."
    ```

Amazon API Gateway stage variables reference

You can use API Gateway stage variables in the following cases.

Parameter mapping expressions

A stage variable can be used in a parameter mapping expression for an API method's request or response header parameter, without any partial substitution. In the following example, the stage variable is referenced without the $ and the enclosing `{...}`.

- `stageVariables.<variable_name>`

Mapping templates

A stage variable can be used anywhere in a mapping template, as shown in the following examples.

- `{ "name" : "$stageVariables.<variable_name>"}`
- `{ "name" : "${stageVariables.<variable_name>}"}`

HTTP integration URIs

A stage variable can be used as part of an HTTP integration URL, as shown in the following examples:

- A full URI without protocol – `http://${stageVariables.<variable_name>}`
- A full domain – `http://${stageVariables.<variable_name>}/resource/operation`
- A subdomain – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- A path – `http://example.com/${stageVariables.<variable_name>}/bar`
- A query string – `http://example.com/foo?q=${stageVariables.<variable_name>}`

AWS integration URIs

A stage variable can be used as part of AWS URI action or path components, as shown in the following example.

- `arn:aws:apigateway:<region>:<service>:${stageVariables.<variable_name>}`

AWS integration URIs (Lambda functions)

A stage variable can be used in place of a Lambda function name, or version/alias, as shown in the following examples.

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

> **Note**
>
> To use a stage variable for a Lambda function, the function must be in the same account as the API. Stage variables don't support cross-account Lambda functions.

AWS integration credentials

A stage variable can be used as part of AWS user/role credential ARN, as shown in the following example.

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

Set up an API Gateway canary release deployment

Canary release is a software development strategy in which a new version of an API (as well as other software) is deployed for testing purposes, and the base version remains deployed as a production release for normal operations on the same stage. For purposes of discussion, we refer to the base version as a production release in this documentation. Although this is reasonable, you are free to apply canary release on any non-production version for testing.

In a canary release deployment, total API traffic is separated at random into a production release and a canary release with a pre-configured ratio. Typically, the canary release receives a small percentage of API traffic and the production release takes up the rest. The updated API features are only visible to API traffic through the canary. You can adjust the canary traffic percentage to optimize test coverage or performance.

By keeping canary traffic small and the selection random, most users are not adversely affected at any time by potential bugs in the new version, and no single user is adversely affected all the time.

After the test metrics pass your requirements, you can promote the canary release to the production release and disable the canary from the deployment. This makes the new features available in the production stage.

Topics

Canary release deployment in API Gateway

In API Gateway, a canary release deployment uses the deployment stage for the production release of the base version of an API, and attaches to the stage a canary release for the new versions, relative to the base version, of the API. The stage is associated with the initial deployment and the canary with subsequent deployments. At the beginning, both the stage and the canary point to the same API version. We use stage and production release interchangeably and use canary and canary release interchangeably throughout this section.

To deploy an API with a canary release, you create a canary release deployment by adding canary settings to the stage of a regular deployment. The canary settings describe the underlying canary release and the stage represents the production release of the API within this deployment. To add canary settings, set `canarySettings` on the deployment stage and specify the following:

- A deployment ID, initially identical to the ID of the base version deployment set on the stage.

- A percentage of API traffic, between 0.0 and 100.0 inclusive, for the canary release.
- Stage variables for the canary release that can override production release stage variables.
- The use of the stage cache for canary requests, if the useStageCache is set and API caching is enabled on the stage.

After a canary release is enabled, the deployment stage cannot be associated with another non-canary release deployment until the canary release is disabled and the canary settings removed from the stage.

When you enable API execution logging, the canary release has its own logs and metrics generated for all canary requests. They are reported to a production stage CloudWatch Logs log group as well as a canary-specific CloudWatch Logs log group. The same applies to access logging. The separate canary-specific logs are helpful to validate new API changes and decide whether to accept the changes and promote the canary release to the production stage, or to discard the changes and revert the canary release from the production stage.

The production stage execution log group is named `API-Gateway-Execution-Logs/{rest-api-id}/{stage-name}` and the canary release execution log group is named `API-Gateway-Execution-Logs/{rest-api-id}/{stage-name}/Canary`. For access logging, you must create a new log group or choose an existing one. The canary release access log group name has the `/Canary` suffix appended to the selected log group name.

A canary release can use the stage cache, if enabled, to store responses and use cached entries to return results to the next canary requests, within a pre-configured time-to-live (TTL) period.

In a canary release deployment, the production release and canary release of the API can be associated with the same version or with different versions. When they are associated with different versions, responses for production and canary requests are cached separately and the stage cache returns corresponding results for production and canary requests. When the production release and canary release are associated with the same deployment, the stage cache uses a single cache key for both types of requests and returns the same response for the same requests from the production release and canary release.

Create a canary release deployment

You create a canary release deployment when deploying the API with canary settings as an additional input to the deployment creation operation.

You can also create a canary release deployment from an existing non-canary deployment by making a `stage:update` request to add the canary settings on the stage.

When creating a non-canary release deployment, you can specify a non-existing stage name. API Gateway creates one if the specified stage does not exist. However, you cannot specify any non-existing stage name when creating a canary release deployment. You will get an error and API Gateway will not create any canary release deployment.

You can create a canary release deployment in API Gateway using the API Gateway console, the AWS CLI, or an AWS SDK.

Topics

Create a canary deployment using the API Gateway console

To use the API Gateway console to create a canary release deployment, follow the instructions below:

To create the initial canary release deployment

1. Sign in to the API Gateway console.
2. Choose an existing REST API or create a new REST API.
3. In the main navigation pane, choose **Resources**, and then choose **Deploy API**. Follow the on-screen instructions in **Deploy API** to deploy the API to a new stage.

   So far, you have deployed the API to a production release stage. Next, you configure canary settings on the stage and, if needed, also enable caching, set stage variables, or configure API execution or access logs.
4. To enable API caching or associate an AWS WAF web ACL with the stage, in the **Stage details** section, choose **Edit**. For more information, see the section called “Cache settings” or the section called “To associate an AWS WAF web ACL with an API Gateway API stage using the API Gateway console”.
5. To configure execution or access logging, in the **Logs and tracing** section, choose **Edit** and follow the on-screen instructions. For more information, see Setting up CloudWatch logging for a REST API in API Gateway.

6. To set stage variables, choose the **Stage variables** tab and follow the on-screen instructions to add or modify stage variables. For more information, see the section called “Set up stage variables”.
7. Choose the **Canary** tab, and then choose **Create canary**. You might need to choose the right arrow button to show the **Canary** tab.
8. Under **Canary settings**, for **Canary**, enter the percentage of requests to be diverted to the canary.
9. If desired, select **Stage cache** to turn on caching for the canary release. The cache is not available for the canary release until API caching is enabled.
10. To override existing stage variables, for **Canary override**, enter a new stage variable value.

After the canary release is initialized on the deployment stage, you change the API and want to test the changes. You can redeploy the API to the same stage so that both the updated version and the base version are accessible through the same stage. The following steps describe how to do that.

To deploy the latest API version to a canary

1. With each update of the API, choose **Deploy API**.
2. In **Deploy API**, choose the stage that contains a canary from the **Deployment stage** dropdown list.
3. (Optional) Enter a description for **Deployment description**.
4. Choose **Deploy** to push the latest API version to the canary release.
5. If desired, reconfigure the stage settings, logs, or canary settings, as describe in To create the initial canary release deployment.

As a result, the canary release points to the latest version while the production release still points to the initial version of the API. The **canarySettings** now has a new **deploymentId** value, whereas the stage still has the initial **deploymentId** value. Behind the scenes, the console calls **stage:update**.

Create a canary deployment using the AWS CLI

First create a baseline deployment with two stage variables, but without any canary:

```
aws apigateway create-deployment \
    --variables sv0=val0,sv1=val1 \
    --rest-api-id abcd1234 \
    --stage-name 'prod' \
```

The command returns a representation of the resulting `Deployment`, similar to the following:

```
{
    "id": "du4ot1",
    "createdDate": 1511379050
}
```

The resulting deployment `id` identifies a snapshot (or version) of the API.

Now create a canary deployment on the `prod` stage:

```
aws apigateway create-deployment  --rest-api-id abcd1234 \
    --canary-settings \
    '{
        "percentTraffic":10.5,
        "useStageCache":false,
        "stageVariableOverrides":{
            "sv1":"val2",
            "sv2":"val3"
        }
    }' \
    --stage-name 'prod'
```

If the specified stage (`prod`) does not exist, the preceding command returns an error. Otherwise, it returns the newly created deployment resource representation similar to the following:

```
{
    "id": "a6rox0",
    "createdDate": 1511379433
}
```

The resulting deployment `id` identifies the test version of the API for the canary release. As a result, the associated stage is canary-enabled. You can view this stage representation by calling the `get-stage` command, similar to the following:

```
aws apigateway get-stage --rest-api-id acbd1234 --stage-name prod
```

The following shows a representation of the `Stage` as the output of the command:

```
{
    "stageName": "prod",
    "variables": {
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "du4ot1",
    "lastUpdatedDate": 1511379433,
    "createdDate": 1511379050,
    "canarySettings": {
        "percentTraffic": 10.5,
        "deploymentId": "a6rox0",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        }
    },
    "methodSettings": {}
}
```

In this example, the base version of the API will use the stage variables of `{"sv0":val0", "sv1":val1"}`, while the test version uses the stage variables of `{"sv1":val2", "sv2":val3"}`. Both the production release and canary release use the same stage variable of `sv1`, but with different values, `val1` and `val2`, respectively. The stage variable of `sv0` is used solely in the production release and the stage variable of `sv2` is used in the canary release only.

You can create a canary release deployment from an existing regular deployment by updating the stage to enable a canary. To demonstrate this, create a regular deployment first:

```
aws apigateway create-deployment \
    --variables sv0=val0,sv1=val1 \
    --rest-api-id abcd1234 \
    --stage-name 'beta'
```

The command returns a representation of the base version deployment:

```
{
    "id": "cifeiw",
    "createdDate": 1511380879
}
```

The associated beta stage does not have any canary settings:

```
{
    "stageName": "beta",
    "variables": {
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "cifeiw",
    "lastUpdatedDate": 1511380879,
    "createdDate": 1511380879,
    "methodSettings": {}
}
```

Now, create a new canary release deployment by attaching a canary on the stage:

```
aws apigateway update-stage \
    --rest-api-id abcd1234 \
    --stage-name 'beta' \
    --patch-operations '[{
            "op": "replace",
            "value": "0.0",
            "path": "/canarySettings/percentTraffic"
        }, {
            "op": "copy",
            "from": "/canarySettings/stageVariableOverrides",
            "path": "/variables"
        }, {
            "op": "copy",
            "from": "/canarySettings/deploymentId",
            "path": "/deploymentId"
        }]'
```

A representation of the updated stage looks like this:

```
{
    "stageName": "beta",
    "variables": {
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "cifeiw",
    "lastUpdatedDate": 1511381930,
    "createdDate": 1511380879,
    "canarySettings": {
        "percentTraffic": 10.5,
        "deploymentId": "cifeiw",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        }
    },
    "methodSettings": {}
}
```

Because we just enabled a canary on an existing version of the API, both the production release (`Stage`) and canary release (`canarySettings`) point to the same deployment, i.e., the same version (`deploymentId`) of the API. After you change the API and deploy it to this stage again, the new version will be in the canary release, while the base version remains in the production release. This is manifested in the stage evolution when the `deploymentId` in the canary release is updated to the new deployment `id` and the `deploymentId` in the production release remains unchanged.

Update a canary release

After a canary release is deployed, you may want to adjust the percentage of the canary traffic or enable or disable the use of a stage cache to optimize the test performance. You can also modify stage variables used in the canary release when the execution context is updated. To make such updates, call the **stage:update** operation with new values on canarySettings.

You can update a canary release using the API Gateway console, the AWS CLI update-stage command or an AWS SDK.

Topics

- Update a canary release using the API Gateway console

- Update a canary release using the AWS CLI

Update a canary release using the API Gateway console

To use the API Gateway console to update existing canary settings on a stage, do the following:

To update existing canary settings

1. Sign in to the API Gateway console and choose an existing REST API.
2. In the main navigation pane, choose **Stages**, and then choose an existing stage.
3. Choose the **Canary** tab, and then choose **Edit**. You might need to choose the right arrow button to show the **Canary** tab.
4. Update the **Request distribution** by increasing or decreasing the percentage number between 0.0 and 100.0, inclusive.
5. Select or clear the **Stage cache** the check box.
6. Add, remove, or modify **Canary stage variables**.
7. Choose **Save**.

Update a canary release using the AWS CLI

To use the AWS CLI to update a canary, call the `update-stage` command.

To enable or disable the use of a stage cache for the canary, call the `update-stage` command as follows:

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}'  \
    --patch-operations op=replace,path=/canarySettings/useStageCache,value=true
```

To adjust the canary traffic percentage, call `update-stage` to replace the `/canarySettings/percentTraffic` value on the stage.

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}' \
    --patch-operations op=replace,path=/canarySettings/percentTraffic,value=25.0
```

To update canary stage variables, including adding, replacing, or removing a canary stage variable:

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}'  \
    --patch-operations '[{
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/newVar",
        "value": "newVal"
      }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/var2",
        "value": "val4"
      }, {
        "op": "remove",
        "path": "/canarySettings/stageVariableOverrides/var1"
      }]'
```

You can update all of the above by combining the operations into a single `patch-operations` value:

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}' \
    --patch-operations '[{
        "op": "replace",
        "path": "/canarySettings/percentTraffic",
        "value": "20.0"
    }, {
        "op": "replace",
        "path": "/canarySettings/useStageCache",
        "value": "true"
    }, {
        "op": "remove",
        "path": "/canarySettings/stageVariableOverrides/var1"
    }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/newVar",
        "value": "newVal"
    }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/val2",
        "value": "val4"
        }]'
```

Promote a canary release

To promote a canary release makes it available in the production stage the API version under testing. The operation involves the following tasks:

- Reset the deployment ID of the stage with the deployment ID settings of the canary. This updates the API snapshot of the stage with the snapshot of the canary, making the test version the production release as well.
- Update stage variables with canary stage variables, if any. This updates the API execution context of the stage with that of the canary. Without this update, the new API version may produce unexpected results if the test version uses different stage variables or different values of existing stage variables.
- Set the percentage of canary traffic to 0.0%.

Promoting a canary release does not disable the canary on the stage. To disable a canary, you must remove the canary settings on the stage.

Topics

Promote a canary release using the API Gateway console

To use the API Gateway console to promote a canary release deployment, do the following:

To promote a canary release deployment

1. Sign in to the API Gateway console and choose an existing API in the primary navigation pane.
2. In the main navigation pane, choose **Stages**, and then choose an existing stage.
3. Choose the **Canary** tab.
4. Choose **Promote canary**.
5. Confirm changes to be made and choose **Promote canary**.

After the promotion, the production release references the same API version (**deploymentId**) as the canary release. You can verify this using the AWS CLI. For example, see the section called "Promote a canary release using the AWS CLI".

Promote a canary release using the AWS CLI

To promote a canary release to the production release using the AWS CLI commands, call the `update-stage` command to copy the canary-associated `deploymentId` to the stage-associated `deploymentId`, to reset the canary traffic percentage to zero (`0.0`), and, to copy any canary-bound stage variables to the corresponding stage-bound ones.

Suppose we have a canary release deployment, described by a stage similar to the following:

```
{
    "_links": {
        ...
    },
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "canarySettings": {
        "deploymentId": "eh1sby",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        },
        "percentTraffic": 10.5
    },
    "createdDate": "2017-11-20T04:42:19Z",
    "deploymentId": "nfcn0x",
    "lastUpdatedDate": "2017-11-22T00:54:28Z",
    "methodSettings": {
        ...
    },
    "stageName": "prod",
    "variables": {
        "sv1": "val1"
    }
}
```

We call the following `update-stage` request to promote it:

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id}  \
    --stage-name '{stage-name}'  \
    --patch-operations '[{
        "op": "replace",
        "value": "0.0",
        "path": "/canarySettings/percentTraffic"
      }, {
        "op": "copy",
        "from": "/canarySettings/stageVariableOverrides",
        "path": "/variables"
      }, {
        "op": "copy",
        "from": "/canarySettings/deploymentId",
        "path": "/deploymentId"
      }]'
```

After the promotion, the stage now looks like this:

```
{
    "_links": {
        ...
    },
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "canarySettings": {
        "deploymentId": "eh1sby",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        },
        "percentTraffic": 0
    },
    "createdDate": "2017-11-20T04:42:19Z",
    "deploymentId": "eh1sby",
    "lastUpdatedDate": "2017-11-22T05:29:47Z",
    "methodSettings": {
        ...
    },
    "stageName": "prod",
    "variables": {
        "sv2": "val3",
        "sv1": "val2"
    }
}
```

As you can see, promoting a canary release to the stage does not disable the canary and the deployment remains to be a canary release deployment. To make it a regular production release deployment, you must disable the canary settings. For more information about how to disable a canary release deployment, see the section called “Turn off a canary release”.

Turn off a canary release

To turn off a canary release deployment is to set the `canarySettings` to null to remove it from the stage.

You can disable a canary release deployment using the API Gateway console, the AWS CLI, or an AWS SDK.

Topics

- Turn off a canary release using the API Gateway console
- Turn off a canary release using the AWS CLI

Turn off a canary release using the API Gateway console

To use the API Gateway console to turn off a canary release deployment, use the following steps:

To turn off a canary release deployment

1. Sign in to the API Gateway console and choose an existing API in the main navigation pane.
2. In the main navigation pane, choose **Stages**, and then choose an existing stage.
3. Choose the **Canary** tab.
4. Choose **Delete**.
5. Confirm you want to delete the canary by choosing **Delete**.

As a result, the canarySettings property becomes null and is removed from the deployment stage. You can verify this using the AWS CLI. For example, see the section called “Turn off a canary release using the AWS CLI”.

Turn off a canary release using the AWS CLI

To use the AWS CLI to turn off a canary release deployment, call the update-stage command as follows:

```
aws apigateway update-stage \
    --rest-api-id abcd1234 \
    --stage-name canary \
    --patch-operations '[{"op":"remove", "path":"/canarySettings"}]'
```

A successful response returns a payload similar to the following:

```
{
    "stageName": "prod",
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "nfcn0x",
    "lastUpdatedDate": 1511309280,
    "createdDate": 1511152939,
    "methodSettings": {
        ...
    }
}
```

As shown in the output, the canarySettings property is no longer present in the stage of a canary-disabled deployment.

Updates to a REST API that require redeployment

Maintaining an API amounts to viewing, updating and deleting the existing API setups. You can maintain an API using the API Gateway console, AWS CLI, an SDK or the API Gateway REST API. Updating an API involves modifying certain resource properties or configuration settings of the API. Resource updates require redeploying the API, whereas configuration updates do not.

API resources that can be updated are detailed in the following table.

API resource updates requiring redeployment of the API

Resource	Remarks
ApiKey	For applicable properties and supported operations, see apikey:update. The update requires redeploying the API.
Authorizer	For applicable properties and supported operations, see authorizer:update. The update requires redeploying the API.
DocumentationPart	For applicable properties and supported operations, see documentationpart:update. The update requires redeploying the API.
DocumentationVersion	For applicable properties and supported operations, see documentationversion:update. The update requires redeploying the API.
GatewayResponse	For applicable properties and supported operations, see gatewayresponse:update. The update requires redeploying the API.
Integration	For applicable properties and supported operations, see integration:update. The update requires redeploying the API.
IntegrationResponse	For applicable properties and supported operations, see integrationresponse:update. The update requires redeploying the API.
Method	For applicable properties and supported operations, see method:update. The update requires redeploying the API.
MethodResponse	For applicable properties and supported operations, see methodresponse:update. The update requires redeploying the API.
Model	For applicable properties and supported operations, see model:update. The update requires redeploying the API.
RequestValidator	For applicable properties and supported operations, see requestvalidator:update. The update requires redeploying the API.
Resource	For applicable properties and supported operations, see resource:update. The update requires redeploying the API.

Resource	Remarks
RestApi	For applicable properties and supported operations, see restapi:update. The update requires redeploying the API.
VpcLink	For applicable properties and supported operations, see vpclink:update. The update requires redeploying the API.

API configurations that can be updated are detailed in the following table.

API configuration updates without requiring redeployment of the API

Configuration	Remarks
Account	For applicable properties and supported operations, see account:update. The update does not require redeploying the API.
Deployment	For applicable properties and supported operations, see deployment:update.
DomainName	For applicable properties and supported operations, see domainname:update. The update does not require redeploying the API.
BasePathMapping	For applicable properties and supported operations, see basepathmapping:update. The update does not require redeploying the API.
Stage	For applicable properties and supported operations, see stage:update. The update does not require redeploying the API.
Usage	For applicable properties and supported operations, see usage:update. The update does not require redeploying the API.
UsagePlan	For applicable properties and supported operations, see usageplan:update. The update does not require redeploying the API.

Setting up custom domain names for REST APIs

Custom domain names are simpler and more intuitive URLs that you can provide to your API users.

After deploying your API, you (and your customers) can invoke the API using the default base URL of the following format:

```
https://api-id.execute-api.region.amazonaws.com/stage
```

where `api-id` is generated by API Gateway, `region` (AWS Region) is specified by you when creating the API, and `stage` is specified by you when deploying the API.

The hostname portion of the URL (that is, `api-id.execute-api.region.amazonaws.com`) refers to an API endpoint. The default API endpoint can be difficult to recall and not user-friendly.

With custom domain names, you can set up your API's hostname, and choose a base path (for example, `myservice`) to map the alternative URL to your API. For example, a more user-friendly API base URL can become:

```
https://api.example.com/myservice
```

Note

A Regional custom domain can be associated with REST APIs and HTTP APIs. You can use API Gateway Version 2 APIs to create and manage Regional custom domain names for REST APIs.
Custom domain names are not supported for private APIs.
You can choose a minimum TLS version that your REST API supports. For REST APIs, you can choose TLS 1.2 or TLS 1.0.

Register a domain name

You must have a registered internet domain name in order to set up custom domain names for your APIs. If needed, you can register an internet domain using Amazon Route 53 or using a third-party domain registrar of your choice. An API's custom domain name can be the name of a subdomain or the root domain (also known as "zone apex") of a registered internet domain.

After a custom domain name is created in API Gateway, you must create or update your DNS provider's resource record to map to your API endpoint. Without such a mapping, API requests bound for the custom domain name cannot reach API Gateway.

Note

An edge-optimized custom domain name is created in a specific Region and owned by a specific AWS account. Moving such a custom domain name between Regions or AWS accounts involves deleting the existing CloudFront distribution and creating a new one. The process may take approximately 30 minutes before the new custom domain name becomes available. For more information, see Updating CloudFront Distributions.

Edge-optimized custom domain names

When you deploy an edge-optimized API, API Gateway sets up an Amazon CloudFront distribution and a DNS record to map the API domain name to the CloudFront distribution domain name. Requests for the API are then routed to API Gateway through the mapped CloudFront distribution.

When you create a custom domain name for an edge-optimized API, API Gateway sets up a CloudFront distribution. But you must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is for API requests that are bound for the custom domain name to be routed to API Gateway through the mapped CloudFront distribution. You must also provide a certificate for the custom domain name.

Note

The CloudFront distribution created by API Gateway is owned by a Region-specific account affiliated with API Gateway. When tracing operations to create and update such a CloudFront distribution in CloudWatch Logs, you must use this API Gateway account ID. For more information, see Log custom domain name creation in CloudTrail.

To set up an edge-optimized custom domain name or to update its certificate, you must have a permission to update CloudFront distributions.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

Create a role for identity federation. Follow the instructions in Creating a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:
  - Create a role that your user can assume. Follow the instructions in Creating a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

The following permissions are required to update CloudFront distributions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCloudFrontUpdateDistribution",
            "Effect": "Allow",
            "Action": [
                "cloudfront:updateDistribution"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

API Gateway supports edge-optimized custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information on using custom domain names on a CloudFront distribution, including the required certificate format and the maximum size of a certificate key length, see Using Alternate Domain Names and HTTPS in the *Amazon CloudFront Developer Guide*.

To set up a custom domain name as your API's hostname, you, as the API owner, must provide an SSL/TLS certificate for the custom domain name.

To provide a certificate for an edge-optimized custom domain name, you can request AWS Certificate Manager (ACM) to generate a new certificate in ACM or to import into ACM one issued by a third-party certificate authority in the `us-east-1` Region (US East (N. Virginia)).

Regional custom domain names

When you create a custom domain name for a Regional API, API Gateway creates a Regional domain name for the API. You must set up a DNS record to map the custom domain name to the Regional domain name. You must also provide a certificate for the custom domain name.

Wildcard custom domain names

With wildcard custom domain names, you can support an almost infinite number of domain names without exceeding the default quota. For example, you could give each of your customers their own domain name, *customername*.api.example.com.

To create a wildcard custom domain name, specify a wildcard (*) as the first subdomain of a custom domain that represents all possible subdomains of a root domain.

For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`, which all route to the same domain.

Wildcard custom domain names support distinct configurations from API Gateway's standard custom domain names. For example, in a single AWS account, you can configure `*.example.com` and `a.example.com` to behave differently.

You can use the `$context.domainName` and `$context.domainPrefix` context variables to determine the domain name that a client used to call your API. To learn more about context variables, see API Gateway mapping template and access logging variable reference.

To create a wildcard custom domain name, you must provide a certificate issued by ACM that has been validated using either the DNS or the email validation method.

> **Note**
>
> You can't create a wildcard custom domain name if a different AWS account has created a custom domain name that conflicts with the wildcard custom domain name. For example, if account A has created `a.example.com`, then account B can't create the wildcard custom domain name `*.example.com`.
> If account A and account B share an owner, you can contact the AWS Support Center to request an exception.

Certificates for custom domain names

⚠ **Important**

You specify the certificate for your custom domain name. If your application uses certificate pinning, sometimes known as SSL pinning, to pin an ACM certificate, the application might not be able to connect to your domain after AWS renews the certificate. For more information, see Certificate pinning problems in the *AWS Certificate Manager User Guide*.

To provide a certificate for a custom domain name in a Region where ACM is supported, you must request a certificate from ACM. To provide a certificate for a Regional custom domain name in a Region where ACM is not supported, you must import a certificate to API Gateway in that Region.

To import an SSL/TLS certificate, you must provide the PEM-formatted SSL/TLS certificate body, its private key, and the certificate chain for the custom domain name. Each certificate stored in ACM is identified by its ARN. To use an AWS managed certificate for a domain name, you simply reference its ARN.

ACM makes it straightforward to set up and use a custom domain name for an API. You create a certificate for the given domain name (or import a certificate), set up the domain name in API Gateway with the ARN of the certificate provided by ACM, and map a base path under the custom domain name to a deployed stage of the API. With certificates issued by ACM, you do not have to worry about exposing any sensitive certificate details, such as the private key.

Topics

Getting certificates ready in AWS Certificate Manager

Before setting up a custom domain name for an API, you must have an SSL/TLS certificate ready in AWS Certificate Manager. The following steps describe how to get this done. For more information, see the AWS Certificate Manager User Guide.

> **Note**
>
> To use an ACM certificate with an API Gateway edge-optimized custom domain name, you must request or import the certificate in the US East (N. Virginia) (`us-east-1`) Region. For an API Gateway Regional custom domain name, you must request or import the certificate in the same Region as your API. The certificate must be signed by a publicly trusted Certificate Authority and cover the custom domain name.

First, register your internet domain, for example, *`example`*`.com`. You can use either Amazon Route 53 or a third-party accredited domain registrar. For a list of such registrars, see Accredited Registrar Directory at the ICANN website.

To create in or import into ACM an SSL/TLS certificate for a domain name, do one of the following:

To request a certificate provided by ACM for a domain name

1. Sign in to the AWS Certificate Manager console.
2. Choose **Request a certificate**.
3. Enter a custom domain name for your API, for example, `api.example.com`, in **Domain name**.
4. Optionally, choose **Add another name to this certificate**.
5. Choose **Review and request**.
6. Choose **Confirm and request**.
7. For a valid request, a registered owner of the internet domain must consent to the request before ACM issues the certificate.

To import into ACM a certificate for a domain name

1. Get a PEM-encoded SSL/TLS certificate for your custom domain name from a certificate authority. For a partial list of such CAs, see the Mozilla Included CA List

a. Generate a private key for the certificate and save the output to a file, using the OpenSSL toolkit at the OpenSSL website:

```
openssl genrsa -out private-key-file 2048
```

> **Note**
>
> Amazon API Gateway leverages Amazon CloudFront to support certificates for custom domain names. As such, the requirements and constraints of a custom domain name SSL/TLS certificate are dictated by CloudFront. For example, the maximum size of the public key is 2048 and the private key size can be 1024, 2048, and 4096. The public key size is determined by the certificate authority you use. Ask your certificate authority to return keys of a size different from the default length. For more information, see Secure access to your objects and Create signed URLs and signed cookies.

b. Generate a certificate signing request (CSR) with the previously generated private key, using OpenSSL:

```
openssl req -new -sha256 -key private-key-file -out CSR-file
```

c. Submit the CSR to the certificate authority and save the resulting certificate.

d. Download the certificate chain from the certificate authority.

> **Note**
>
> If you obtain the private key in another way and the key is encrypted, you can use the following command to decrypt the key before submitting it to API Gateway for setting up a custom domain name.
>
> ```
> openssl pkcs8 -topk8 -inform pem -in MyEncryptedKey.pem -outform pem -
> nocrypt -out MyDecryptedKey.pem
> ```

2. Upload the certificate to AWS Certificate Manager:

   a. Sign in to the AWS Certificate Manager console.

b. Choose **Import a certificate**.

c. For **Certificate body**, enter or paste the body of the PEM-formatted server certificate from your certificate authority. The following shows an abbreviated example of such a certificate.

```
-----BEGIN CERTIFICATE-----
EXAMPLECA+KgAwIBAgIQJ1XxJ8Pl++gOfQtj0IBoqDANBgkqhkiG9w0BAQUFADBB
...
az8Cg1aicxLBQ7EaWIhhgEXAMPLE
-----END CERTIFICATE-----
```

d. For **Certificate private key**, enter or paste your PEM-formatted certificate's private key. The following shows an abbreviated example of such a key.

```
-----BEGIN RSA PRIVATE KEY-----
EXAMPLEBAAKCAQEA2Qb3LDHD7StY7Wj6U2/opV6Xu37qUCCkeDWhwpZMYJ9/nETO
...
1qGvJ3u04vdnzaYN5WoyN5LFckrlA71+CszD1CGSqbVDWEXAMPLE
-----END RSA PRIVATE KEY-----
```

e. For **Certificate chain**, enter or paste the PEM-formatted intermediate certificates and, optionally, the root certificate, one after the other without any blank lines. If you include the root certificate, your certificate chain must start with intermediate certificates and end with the root certificate. Use the intermediate certificates provided by your certificate authority. Do not include any intermediaries that are not in the chain of trust path. The following shows an abbreviated example.

```
-----BEGIN CERTIFICATE-----
EXAMPLECA4ugAwIBAgIQWrYdrB5NogYUx1U9Pamy3DANBgkqhkiG9w0BAQUFADCB
...
8/ifBlIK3se2e4/hEfcEejX/arxbx1BJCHBvlEPNnsdw8EXAMPLE
-----END CERTIFICATE-----
```

Here is another example.

```
-----BEGIN CERTIFICATE-----
Intermediate certificate 2
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
Intermediate certificate 1
```

```
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
Optional: Root certificate
-----END CERTIFICATE-----
```

f. Choose **Review and import**.

After the certificate is successfully created or imported, make note of the certificate ARN. You need it when setting up the custom domain name.

Choosing a security policy for your custom domain in API Gateway

For greater security of your Amazon API Gateway custom domain, you can choose a security policy in the API Gateway console, the AWS CLI, or an AWS SDK.

A *security policy* is a predefined combination of minimum TLS version and cipher suites offered by API Gateway. You can choose either a TLS version 1.2 or TLS version 1.0 security policy. The TLS protocol addresses network security problems such as tampering and eavesdropping between a client and server. When your clients establish a TLS handshake to your API through the custom domain, the security policy enforces the TLS version and cipher suite options your clients can choose to use.

In custom domain settings, a security policy determines two settings:

- The minimum TLS version that API Gateway uses to communicate with API clients
- The cipher that API Gateway uses to encrypt the content that it returns to API clients

If you choose a TLS 1.0 security policy, the security policy accepts TLS 1.0, TLS 1.2, and TLS 1.3 traffic. If you choose a TLS 1.2 security policy, the security policy accepts TLS 1.2 and TLS 1.3 traffic and rejects TLS 1.0 traffic.

> **Note**
>
> You can only specify a security policy for a custom domain. For an API using a default endpoint, API Gateway uses the following security policy:
>
> - For edge-optimized APIs: `TLS-1-0`
> - For Regional APIs: `TLS-1-0`

- For private APIs: TLS-1-2

Topics

How to specify a security policy for custom domains

When you create a custom domain name, you specify the security policy for it. To learn how to create a custom domain, see the section called "Creating an edge-optimized custom domain name" or the section called "Setting up a regional custom domain name".

To change the security policy of your custom domain name, update the custom domain settings. You can update your custom domain name settings using the AWS Management Console, the AWS CLI, or an AWS SDK.

When you use the API Gateway REST API or AWS CLI, specify the new TLS version, `TLS_1_0` or `TLS_1_2` in the `securityPolicy` parameter. For more information, see domainname:update in the *Amazon API Gateway REST API Reference* or update-domain-name in the *AWS CLI Reference*.

The update operation may take few minutes to complete.

Supported security policies, TLS protocol versions, and ciphers for edge-optimized custom domains

The following table describes the security policies that can be specified for edge-optimized custom domain names.

Security policy	TLS_1_0	TLS_1_2
TLS protocols		

Security policy	TLS_1_0	TLS_1_2
TLSv1.3	♦	♦
TLSv1.2	♦	♦
TLSv1.1	♦	
TLSv1	♦	
TLS ciphers		
TLS_AES_128_GCM_SHA256	♦	♦
TLS_AES_256_GCM_SHA384	♦	♦
TLS_CHACHA20_POLY1305_SHA256	♦	♦
ECDHE-ECDSA-AES128-GCM-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA	♦	
ECDHE-ECDSA-AES256-GCM-SHA384	♦	♦
ECDHE-ECDSA-CHACHA20-POLY1305	♦	♦
ECDHE-ECDSA-AES256-SHA384	♦	♦
ECDHE-ECDSA-AES256-SHA	♦	
ECDHE-RSA-AES128-GCM-SHA256	♦	♦

Security policy	TLS_1_0	TLS_1_2
ECDHE-RSA-AES128-SHA256	♦	♦
ECDHE-RSA-AES128-SHA	♦	
ECDHE-RSA-AES256-GCM-SHA384	♦	♦
ECDHE-RSA-CHACHA20-POLY1305	♦	♦
ECDHE-RSA-AES256-SHA384	♦	♦
ECDHE-RSA-AES256-SHA	♦	
AES128-GCM-SHA256	♦	
AES256-GCM-SHA384	♦	♦
AES128-SHA256	♦	♦
AES256-SHA	♦	
AES128-SHA	♦	
DES-CBC3-SHA	♦	

Supported security policies, TLS protocol versions, and ciphers for Regional custom domains

The following table describes the security policies that can be specified for Regional custom domain names.

Security policy	TLS_1_0	TLS_1_2
TLS protocols		
TLSv1.3	♦	♦
TLSv1.2	♦	♦

Security policy	TLS_1_0	TLS_1_2
TLSv1.1	♦	
TLSv1	♦	
TLS ciphers		
TLS_AES_128_GCM_SHA256	♦	♦
TLS_AES_256_GCM_SHA384	♦	♦
TLS_CHACHA20_POLY1305_SHA256	♦	♦
ECDHE-ECDSA-AES128-GCM-SHA256	♦	♦
ECDHE-RSA-AES128-GCM-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA256	♦	♦
ECDHE-RSA-AES128-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA	♦	
ECDHE-RSA-AES128-SHA	♦	
ECDHE-ECDSA-AES256-GCM-SHA384	♦	♦
ECDHE-RSA-AES256-GCM-SHA384	♦	♦
ECDHE-ECDSA-AES256-SHA384	♦	♦
ECDHE-RSA-AES256-SHA384	♦	♦

Security policy	TLS_1_0	TLS_1_2
ECDHE-RSA-AES256-SHA	♦	
ECDHE-ECDSA-AES256-SHA	♦	
AES128-GCM-SHA256	♦	♦
AES128-SHA256	♦	♦
AES128-SHA	♦	
AES256-GCM-SHA384	♦	♦
AES256-SHA256	♦	♦
AES256-SHA	♦	

Supported TLS protocol versions and ciphers for private APIs

The following table describes the supported TLS protocol and ciphers for private APIs. Specifying a security policy for private APIs is not supported.

Security policy	TLS_1_2
TLS protocols	
TLSv1.2	♦
TLS ciphers	
ECDHE-ECDSA-AES128-GCM-SHA256	♦
ECDHE-RSA-AES128-GCM-SHA256	♦
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256-GCM-SHA384	♦

Security policy	TLS_1_2
ECDHE-RSA-AES256-GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL and RFC cipher names

OpenSSL and IETF RFC 5246 use different names for the same ciphers. The following table maps the OpenSSL name to the RFC name for each cipher.

OpenSSL cipher name	RFC cipher name
TLS_AES_128_GCM_SHA256	TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256
ECDHE-RSA-AES128-GCM-SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

OpenSSL cipher name	RFC cipher name
ECDHE-RSA-AES128-SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
ECDHE-RSA-AES128-SHA	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
ECDHE-RSA-AES256-GCM-SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
ECDHE-RSA-AES256-SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
ECDHE-RSA-AES256-SHA	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
AES128-GCM-SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256
AES256-GCM-SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384
AES128-SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256
AES256-SHA	TLS_RSA_WITH_AES_256_CBC_SHA
AES128-SHA	TLS_RSA_WITH_AES_128_CBC_SHA
DES-CBC3-SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA

Information about HTTP APIs and WebSocket APIs

For more information about HTTP APIs and WebSocket APIs, see the section called “Security policy for HTTP APIs” and the section called “Security policy for WebSocket APIs”.

Creating an edge-optimized custom domain name

Topics

- Set up an edge-optimized custom domain name for an API Gateway API
- Log custom domain name creation in CloudTrail
- Configure base path mapping of an API with a custom domain name as its hostname
- Rotate a certificate imported into ACM
- Call your API with custom domain names

Set up an edge-optimized custom domain name for an API Gateway API

The following procedure describes how to create a custom domain name for an API using the API Gateway console.

To create a custom domain name using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Custom domain names** from the main navigation pane.
3. Choose **Create**.
4. For **Domain name**, enter a domain name.
5. Under Configuration, choose **Edge-optimized**.
6. Choose a minimum TLS version.
7. Choose an ACM certificate.

   > **Note**
   >
   > To use an ACM certificate with an API Gateway edge-optimized custom domain name, you must request or import the certificate in the `us-east-1` Region (US East (N. Virginia)).

8. Choose **Create domain name**.
9. After the custom domain name is created, the console displays the associated CloudFront distribution domain name, in the form of *`distribution-id`*`.cloudfront.net`, along with the certificate ARN. Note the CloudFront distribution domain name shown in the output. You

need it in the next step to set the custom domain's CNAME value or A-record alias target in your DNS.

Note

The newly created custom domain name takes about 40 minutes to be ready. In the meantime, you can configure the DNS record alias to map the custom domain name to the associated CloudFront distribution domain name and to set up the base path mapping for the custom domain name while the custom domain name is being initialized.

10. Next, you configure DNS records with your DNS provider to map the custom domain name to the associated CloudFront distribution. For instructions for Amazon Route 53, see Routing traffic to an Amazon API Gateway API by using your domain name in the *Amazon Route 53 Developer Guide*.

    For most DNS providers, a custom domain name is added to the hosted zone as a CNAME resource record set. The CNAME record name specifies the custom domain name you entered earlier in **Domain Name** (for example, `api.example.com`). The CNAME record value specifies the domain name for the CloudFront distribution. However, use of a CNAME record will not work if your custom domain is a zone apex (i.e., `example.com` instead of `api.example.com`). A zone apex is also commonly known as the root domain of your organization. For a zone apex you need to use an A-record alias, provided that is supported by your DNS provider.

    With Route 53 you can create an A record alias for your custom domain name and specify the CloudFront distribution domain name as the alias target. This means that Route 53 can route your custom domain name even if it is a zone apex. For more information, see Choosing Between Alias and Non-Alias Resource Record Sets in the *Amazon Route 53 Developer Guide*.

    Use of A-record aliases also eliminates exposure of the underlying CloudFront distribution domain name because the domain name mapping takes place solely within Route 53. For these reasons, we recommend that you use Route 53 A-record alias whenever possible.

In addition to using the API Gateway console, you can use the API Gateway REST API, AWS CLI or one of the AWS SDKs to set up the custom domain name for your APIs. As an illustration, the following procedure outlines the steps to do so using the REST API calls.

To set up a custom domain name using the API Gateway REST API

1. Call domainname:create, specifying the custom domain name and the ARN of a certificate stored in AWS Certificate Manager.

   The successful API call returns a `201 Created` response containing the certificate ARN as well as the associated CloudFront distribution name in its payload.
2. Note the CloudFront distribution domain name shown in the output. You need it in the next step to set the custom domain's CNAME value or A-record alias target in your DNS.
3. Follow the previous procedure to set up an A-record alias to map the custom domain name to its CloudFront distribution name.

For code examples of this REST API call, see domainname:create.

Log custom domain name creation in CloudTrail

When CloudTrail is enabled for logging API Gateway calls made by your account, API Gateway logs the associated CloudFront distribution updates when a custom domain name is created or updated for an API. Because these CloudFront distributions are owned by API Gateway, each of these reported CloudFront distributions is identified by one of the following Region-specific API Gateway account IDs, instead of the API owner's account ID.

Region	Account ID
us-east-1	392220576650
us-east-2	718770453195
us-west-1	968246515281
us-west-2	109351309407
ca-central-1	796887884028
eu-west-1	631144002099
eu-west-2	544388816663
eu-west-3	061510835048

Region	Account ID
eu-central-1	474240146802
eu-central-2	166639821150
eu-north-1	394634713161
eu-south-1	753362059629
eu-south-2	359345898052
ap-northeast-1	969236854626
ap-northeast-2	020402002396
ap-northeast-3	360671645888
ap-southeast-1	195145609632
ap-southeast-2	798376113853
ap-southeast-3	652364314486
ap-southeast-4	849137399833
ap-south-1	507069717855
ap-south-2	644042651268
ap-east-1	174803364771
sa-east-1	287228555773
me-south-1	855739686837
me-central-1	614065512851

Configure base path mapping of an API with a custom domain name as its hostname

You can use a single custom domain name as the hostname of multiple APIs. You achieve this by configuring the base path mappings on the custom domain name. With the base path mappings, an API under the custom domain is accessible through the combination of the custom domain name and the associated base path.

For example, if you created an API named `PetStore` and another API named `PetShop` and set up a custom domain name of `api.example.com` in API Gateway, you can set the `PetStore` API's URL as `https://api.example.com` or `https://api.example.com/myPetStore`. The `PetStore` API is associated with the base path of an empty string or `myPetStore` under the custom domain name of `api.example.com`. Similarly, you can assign a base path of `yourPetShop` for the `PetShop` API. The URL of `https://api.example.com/yourPetShop` is then the root URL of the `PetShop` API.

Before setting the base path for an API, complete the steps in Set up an edge-optimized custom domain name for an API Gateway API.

The following procedure sets up API mappings to map paths from your custom domain name to your API stages.

To create API mappings name using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a custom domain name.
3. Choose **Configure API mappings**.
4. Choose **Add new mapping**.
5. Specify the **API**, **Stage**, and **Path** (optional) for the mapping.
6. Choose **Save**.

In addition, you can call the API Gateway REST API, AWS CLI, or one of the AWS SDKs to set up the base path mapping of an API with a custom domain name as its hostname. As an illustration, the following procedure outlines the steps to do so using the REST API calls.

To set up the base path mapping of an API using the API Gateway REST API

- Call basepathmapping:create on a specific custom domain name, specifying the `basePath`, `restApiId`, and a deployment `stage` property in the request payload.

The successful API call returns a `201 Created` response.

For code examples of the REST API call, see basepathmapping:create.

Rotate a certificate imported into ACM

ACM automatically handles renewal of certificates it issues. You do not need to rotate any ACM-issued certificates for your custom domain names. CloudFront handles it on your behalf.

However, if you import a certificate into ACM and use it for a custom domain name, you must rotate the certificate before it expires. This involves importing a new third-party certificate for the domain name and rotate the existing certificate to the new one. You need to repeat the process when the newly imported certificate expires. Alternatively, you can request ACM to issue a new certificate for the domain name and rotate the existing one to the new ACM-issued certificate. After that, you can leave ACM and CloudFront to handle the certificate rotation for you automatically. To create or import a new ACM certificate, follow the steps to request or import a new ACM certificate for the specified domain name.

To rotate a certificate for a domain name, you can use the API Gateway console, the API Gateway REST API, AWS CLI, or one of the AWS SDKs.

To rotate an expiring certificate imported into ACM using the API Gateway console

1. Request or import a certificate in ACM.
2. Go back to the API Gateway console.
3. Choose **Custom domain names** from the API Gateway console main navigation pane.
4. Choose a custom domain name.
5. Choose **Edit**.
6. Choose the desired certificate from the **ACM certificate** dropdown list.
7. Choose **Save** to begin rotating the certificate for the custom domain name.

> **Note**
>
> It takes about 40 minutes for the process to finish. After the rotation is done, you can choose the two-way arrow icon next to **ACM Certificate** to roll back to the original certificate.

To illustrate how to programmatically rotate an imported certificate for a custom domain name, we outline the steps using the API Gateway REST API.

Rotate an imported certificate using the API Gateway REST API

- Call domainname:update action, specifying the ARN of the new ACM certificate for the specified domain name.

Call your API with custom domain names

Calling an API with a custom domain name is the same as calling the API with its default domain name, provided that the correct URL is used.

The following examples compare and contrast a set of default URLs and corresponding custom URLs of two APIs (`udxjef` and `qf3duz`) in a specified Region (`us-east-1`), and of a given custom domain name (`api.example.com`).

Root URLs of APIs with default and custom domain names

API ID	Stage	Default URL	Base path	Custom URL
udxjef	prod	https://udxjef.execute-api.us-east-1.amazonaws.com/prod	/petstore	https://api.example.com/petstore
udxjef	tst	https://udxjef.execute-api.us-east-1.amazonaws.com/tst	/petdepot	https://api.example.com/petdepot
qf3duz	dev	https://qf3duz.execute-api.us-east-1.amazonaws.com/dev	/bookstore	https://api.example.com/bookstore

API ID	Stage	Default URL	Base path	Custom URL
qf3duz	tst	https://qf3duz.execute-api.us-east-1.amazonaws.com/tst	/bookstand	https://api.example.com/bookstand

API Gateway supports custom domain names for an API by using Server Name Indication (SNI). You can invoke the API with a custom domain name using a browser or a client library that supports SNI.

API Gateway enforces SNI on the CloudFront distribution. For information on how CloudFront uses custom domain names, see Amazon CloudFront Custom SSL.

Setting up a regional custom domain name in API Gateway

You can create a custom domain name for a Regional API endpoint (for an AWS Region). To create a custom domain name, you must provide a Region-specific ACM certificate. For more information about creating or uploading a custom domain name certificate, see Getting certificates ready in AWS Certificate Manager.

⚠ **Important**

For an API Gateway Regional custom domain name, you must request or import the certificate in the same Region as your API.

When you create a Regional custom domain name (or migrate one) with an ACM certificate, API Gateway creates a service-linked role in your account if the role doesn't exist already. The service-linked role is required to attach your ACM certificate to your Regional endpoint. The role is named **AWSServiceRoleForAPIGateway** and will have the **APIGatewayServiceRolePolicy** managed policy attached to it. For more information about use of the service-linked role, see Using Service-Linked Roles.

> ⚠ **Important**
>
> You must create a DNS record to point the custom domain name to the Regional domain name. This enables the traffic that is bound to the custom domain name to be routed to the API's Regional hostname. The DNS record can be the CNAME or "A" type.

Topics

- [Set up a regional custom domain name with an ACM certificate using the API Gateway console]
- [Set up a regional custom domain name with an ACM certificate using AWS CLI]

Set up a regional custom domain name with an ACM certificate using the API Gateway console

To use the API Gateway console to set up a Regional custom domain name, use the following procedure.

To set up a regional custom domain name using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Custom domain names** from the main navigation pane.
3. Choose **Create**.
4. For **Domain name**, enter a domain name.
5. Under Configuration, choose **Regional**.
6. Choose a minimum TLS version.
7. Choose an ACM certificate. The certificate must be in the same Region as the API.
8. Choose **Create**.
9. Follow the Route 53 documentation on configuring Route 53 to route traffic to API Gateway.

The following procedure sets up API mappings to map paths from your custom domain name to your API stages.

To create API mappings name using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a custom domain name.

3. Choose **Configure API mappings**.

4. Choose **Add new mapping**.

5. Specify the **API**, **Stage**, and **Path** for the mapping.

6. Choose **Save**.

To learn about setting basepath mappings for the custom domain, see Configure base path mapping of an API with a custom domain name as its hostname.

Set up a regional custom domain name with an ACM certificate using AWS CLI

To use the AWS CLI to set up a custom domain name for a Regional API, use the following procedure.

1. Call `create-domain-name`, specifying a custom domain name and the ARN of a Regional certificate.

```
aws apigatewayv2 create-domain-name \
    --domain-name 'regional.example.com' \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678
```

   Note that the specified certificate is from the `us-west-2` Region and for this example, we assume that the underlying API is from the same Region.

   If successful, the call returns a result similar to the following:

```
{
    "ApiMappingSelectionExpression": "$request.basepath",
    "DomainName": "regional.example.com",
    "DomainNameConfigurations": [
        {
            "ApiGatewayDomainName": "d-id.execute-api.us-west-2.amazonaws.com",
            "CertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/id",
            "DomainNameStatus": "AVAILABLE",
            "EndpointType": "REGIONAL",
            "HostedZoneId": "id",
            "SecurityPolicy": "TLS_1_2"
        }
    ]
}
```

The `DomainNameConfigurations` property value returns the Regional API's hostname. You must create a DNS record to point your custom domain name to this Regional domain name. This enables the traffic that is bound to the custom domain name to be routed to this Regional API's hostname.

2. Create a DNS record to associate the custom domain name and the Regional domain name. This enables requests that are bound to the custom domain name to be routed to the API's Regional hostname.

3. Add a base path mapping to expose the specified API (for example, `0qzs2sy7bh`) in a deployment stage (for example, `test`) under the specified custom domain name (for example, `regional.example.com`).

   ```
   aws apigatewayv2 create-api-mapping \
       --domain-name 'regional.example.com' \
       --api-mapping-key 'myApi' \
       --api-id 0qzs2sy7bh \
       --stage 'test'
   ```

   As a result, the base URL using the custom domain name for the API that is deployed in the stage becomes `https://regional.example.com/myAPI`.

4. Configure your DNS records to map the Regional custom domain name to its hostname of the given hosted zone ID. First create a JSON file that contains the configuration for setting up a DNS record for the Regional domain name. The following example shows how to create a DNS A record to map a Regional custom domain name (`regional.example.com`) to its Regional hostname (`d-numh1z56v6.execute-api.us-west-2.amazonaws.com`) provisioned as part of the custom domain name creation. The `DNSName` and `HostedZoneId` properties of `AliasTarget` can take the `regionalDomainName` and `regionalHostedZoneId` values, respectively, of the custom domain name. You can also get the Regional Route 53 Hosted Zone IDs in Amazon API Gateway Endpoints and Quotas.

   ```
   {
     "Changes": [
       {
         "Action": "CREATE",
         "ResourceRecordSet": {
           "Name": "regional.example.com",
           "Type": "A",
   ```

```
        "AliasTarget": {
          "DNSName": "d-numh1z56v6.execute-api.us-west-2.amazonaws.com",
          "HostedZoneId": "Z2OJLYMUO9EFXC",
          "EvaluateTargetHealth": false
        }
      }
    }
  ]
}
```

5. Run the following CLI command:

```
aws route53 change-resource-record-sets \
    --hosted-zone-id {your-hosted-zone-id} \
    --change-batch file://path/to/your/setup-dns-record.json
```

   where *{your-hosted-zone-id}* is the Route 53 Hosted Zone ID of the DNS record set in your account. The `change-batch` parameter value points to a JSON file (*setup-dns-record.json*) in a folder (*path/to/your*).

Migrating a custom domain name to a different API endpoint

You can migrate your custom domain name between edge-optimized and Regional endpoints. You first add the new endpoint configuration type to the existing `endpointConfiguration.types` list for the custom domain name. Next, you set up a DNS record to point the custom domain name to the newly provisioned endpoint. An optional last step is to remove the obsolete custom domain name configuration data.

When planning the migration, remember that for an edge-optimized API's custom domain name, the required certificate provided by ACM must be from the US East (N. Virginia) Region (`us-east-1`). This certificate is distributed to all the geographic locations. However, for a Regional API, the ACM certificate for the Regional domain name must be from the same Region hosting the API. You can migrate an edge-optimized custom domain name that is not in the `us-east-1` Region to a Regional custom domain name by first requesting a new ACM certificate from the Region that is local to the API.

It may take up to 60 seconds to complete a migration between an edge-optimized custom domain name and a Regional custom domain name in API Gateway. For the newly created endpoint to become ready to accept traffic, the migration time also depends on when you update your DNS records.

Topics

- Migrate custom domain names using the AWS CLI

Migrate custom domain names using the AWS CLI

To use the AWS CLI to migrate a custom domain name from an edge-optimized endpoint to a Regional endpoint or vice versa, call the `update-domain-name` command to add the new endpoint type and, optionally, call the `update-domain-name` command to remove the old endpoint type.

Topics

- Migrate an edge-optimized custom domain name to regional
- Migrate a regional custom domain name to edge-optimized

Migrate an edge-optimized custom domain name to regional

To migrate an edge-optimized custom domain name to a Regional custom domain name, call the `update-domain-name` CLI command, as follows:

```
aws apigateway update-domain-name \
    --domain-name 'api.example.com' \
    --patch-operations  [ \
        { op:'add', path: '/endpointConfiguration/types',value: 'REGIONAL' }, \
        { op:'add', path: '/regionalCertificateArn', value: 'arn:aws:acm:us-
west-2:123456789012:certificate/cd833b28-58d2-407e-83e9-dce3fd852149' } \
      ]
```

The Regional certificate must be of the same Region as the Regional API.

The success response has a `200 OK` status code and a body similar to the following:

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:123456789012:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "api.example.com",
    "endpointConfiguration": {
        "types": [
            "EDGE",
            "REGIONAL"
        ]
    },
    "regionalCertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/
cd833b28-58d2-407e-83e9-dce3fd852149",
    "regionalDomainName": "d-fdisjghyn6.execute-api.us-west-2.amazonaws.com"
}
```

For the migrated Regional custom domain name, the resulting `regionalDomainName` property returns the Regional API hostname. You must set up a DNS record to point the Regional custom domain name to this Regional hostname. This enables the traffic that is bound to the custom domain name to be routed to the Regional host.

After the DNS record is set, you can remove the edge-optimized custom domain name by calling the update-domain-name command of AWS CLI:

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations [ \
            {op:'remove', path:'/endpointConfiguration/types', value:'EDGE'}, \
            {op:'remove', path:'certificateName'}, \
            {op:'remove', path:'certificateArn'} \
        ]
```

Migrate a regional custom domain name to edge-optimized

To migrate a Regional custom domain name to an edge-optimized custom domain name, call the `update-domain-name` command of the AWS CLI, as follows:

```
aws apigateway update-domain-name \
    --domain-name 'api.example.com' \
    --patch-operations  [ \
            { op:'add', path:'/endpointConfiguration/types',value: 'EDGE' }, \
            { op:'add', path:'/certificateName', value:'edge-cert'}, \
            { op:'add', path:'/certificateArn', value: 'arn:aws:acm:us-
east-1:123456789012:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a' } \
        ]
```

The edge-optimized domain certificate must be created in the `us-east-1` Region.

The success response has a `200 OK` status code and a body similar to the following:

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:738575810317:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "api.example.com",
    "endpointConfiguration": {
        "types": [
            "EDGE",
            "REGIONAL"
        ]
    },
    "regionalCertificateArn": "arn:aws:acm:us-
east-1:123456789012:certificate/3d881b54-851a-478a-a887-f6502760461d",
    "regionalDomainName": "d-cgkq2qwgzf.execute-api.us-east-1.amazonaws.com"
}
```

For the specified custom domain name, API Gateway returns the edge-optimized API hostname as the `distributionDomainName` property value. You must set a DNS record to point the edge-optimized custom domain name to this distribution domain name. This enables traffic that is bound to the edge-optimized custom domain name to be routed to the edge-optimized API hostname.

After the DNS record is set, you can remove the `REGION` endpoint type of the custom domain name:

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations [ \
        {op:'remove', path:'/endpointConfiguration/types', value:'REGIONAL'}, \
        {op:'remove', path:'regionalCertificateArn'} \
      ]
```

The result of this command is similar to the following output, with only edge-optimized domain name configuration data:

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:738575810317:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "regional.haymuto.com",
    "endpointConfiguration": {
        "types": "EDGE"
    }
}
```

Working with API mappings for REST APIs

You use API mappings to connect API stages to a custom domain name. After you create a domain name and configure DNS records, you use API mappings to send traffic to your APIs through your custom domain name.

An API mapping specifies an API, a stage, and optionally a path to use for the mapping. For example, you can map the `production` stage of an API to `https://api.example.com/orders`.

You can map HTTP and REST API stages to the same custom domain name.

Before you create an API mapping, you must have an API, a stage, and a custom domain name. To learn more about creating a custom domain name, see the section called “Setting up a regional custom domain name”.

Routing API requests

You can configure API mappings with multiple levels, for example `orders/v1/items` and `orders/v2/items`.

> **Note**
>
> To configure API mappings with multiple levels, your custom domain name must be regional and use the TLS 1.2 security policy.

For API mappings with multiple levels, API Gateway routes requests to the API mapping that has the longest matching path. API Gateway considers only the paths configured for API mappings, and not API routes, to select the API to invoke. If no path matches the request, API Gateway sends the request to the API that you've mapped to the empty path `(none)`.

For custom domain names that use API mappings with multiple levels, API Gateway routes requests to the API mapping that has the longest matching prefix.

For example, consider a custom domain name `https://api.example.com` with the following API mappings:

1. `(none)` mapped to API 1.
2. `orders` mapped to API 2.
3. `orders/v1/items` mapped to API 3.
4. `orders/v2/items` mapped to API 4.
5. `orders/v2/items/categories` mapped to API 5.

Request	Selected API	Explanation
`https://api.example.com/orders`	`API 2`	The request exactly matches this API mapping.
`https://api.example.com/orders/v1/items`	`API 3`	The request exactly matches this API mapping.
`https://api.example.com/orders/v2/items`	`API 4`	The request exactly matches this API mapping.
`https://api.example.com/orders/v1/items/123`	`API 3`	API Gateway chooses the mapping that has the longest matching path. The 123 at the end of the request doesn't affect the selection.
`https://api.example.com/orders/v2/items/categories/5`	`API 5`	API Gateway chooses the mapping that has the longest matching path.

Request	Selected API	Explanation
`https://api.example.com/customers`	`API 1`	API Gateway uses the empty mapping as a catch-all.
`https://api.example.com/ordersandmore`	`API 2`	API Gateway chooses the mapping that has the longest matching prefix. For a custom domain name configured with single-level mappings, such as only `https://api.example.com/orders` and `https://api.example.com/` , API Gateway would choose `API 1`, as there is no matching path with `ordersandmore` .

Restrictions

- In an API mapping, the custom domain name and mapped APIs must be in the same AWS account.
- API mappings must contain only letters, numbers, and the following characters: `$-_.+!*'()/`.
- The maximum length for the path in an API mapping is 300 characters.
- You can have 200 API mappings with multiple levels for each domain name.
- You can only map HTTP APIs to a regional custom domain name with the TLS 1.2 security policy.
- You can't map WebSocket APIs to the same custom domain name as an HTTP API or REST API.

Create an API mapping

To create an API mapping, you must first create a custom domain name, API, and stage. For information about creating a custom domain name, see the section called "Setting up a regional custom domain name".

For example AWS Serverless Application Model templates that create all resources, see Sessions With SAM on GitHub.

AWS Management Console

To create an API mapping

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Custom domain names**.
3. Select a custom domain name that you've already created.
4. Choose **API mappings**.
5. Choose **Configure API mappings**.
6. Choose **Add new mapping**.
7. Enter an **API**, a **Stage**, and optionally a **Path**.
8. Choose **Save**.

AWS CLI

The following AWS CLI command creates an API mapping. In this example, API Gateway sends requests to `api.example.com/v1/orders` to the specified API and stage.

> **Note**
>
> To create API mappings with multiple levels, you must use `apigatewayv2`.

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1/orders \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

The following AWS CloudFormation example creates an API mapping.

> **Note**
>
> To create API mappings with multiple levels, you must use `AWS::ApiGatewayV2`.

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'orders/v2/items'
    ApiId: !Ref MyApi
    Stage: !Ref MyStage
```

Disabling the default endpoint for a REST API

By default, clients can invoke your API by using the `execute-api` endpoint that API Gateway generates for your API. To ensure that clients can access your API only by using a custom domain name, disable the default `execute-api` endpoint. Clients can still connect to your default endpoint, but they will receive a `403 Forbidden` status code.

> **Note**
>
> When you disable the default endpoint, it affects all stages of an API.

The following AWS CLI command disables the default endpoint for a REST API.

```
aws apigateway update-rest-api \
    --rest-api-id abcdef123 \
    --patch-operations op=replace,path=/disableExecuteApiEndpoint,value='True'
```

After you disable the default endpoint, you must deploy your API for the change to take effect.

The following AWS CLI command creates a deployment.

```
aws apigateway create-deployment \
    --rest-api-id abcdef123 \
    --stage-name dev
```

Configure custom health checks for DNS failover

You can use Amazon Route 53 health checks to control DNS failover from an API Gateway API in a primary AWS Region to one in a secondary Region. This can help mitigate impacts in the event of a Regional issue. If you use a custom domain, you can perform failover without requiring clients to change API endpoints.

When you choose Evaluate Target Health for an alias record, those records fail only when the API Gateway service is unavailable in the Region. In some cases, your own API Gateway APIs can experience interruption before that time. To control DNS failover directly, configure custom Route 53 health checks for your API Gateway APIs. For this example, you use a CloudWatch alarm that helps operators control DNS failover. For more examples and other considerations when you configure failover, see Creating Disaster Recovery Mechanisms Using Route 53 and Performing Route 53 health checks on private resources in a VPC with AWS Lambda and CloudWatch.

Topics

Prerequisites

To complete this procedure, you must create and configure the following resources:

- A domain name that you own.
- An ACM certificate for that domain name in two AWS Regions. For more info, see the section called “Getting certificates ready in AWS Certificate Manager”.
- A Route 53 hosted zone for your domain name. For more information, see Working with hosted zones in the Amazon Route 53 Developer Guide.

For more information on how to create Route 53 failover DNS records for the domain names, see Choose a routing policy in the Amazon Route 53 Developer Guide. For more information on how to monitor a CloudWatch alarm, see Monitoring a CloudWatch alarm in the Amazon Route 53 Developer Guide.

Step 1: Set up resources

In this example, you create the following resources to configure DNS failover for your domain name:

- API Gateway APIs in two AWS Regions
- API Gateway custom domain names with the same name in two AWS Regions
- API Gateway API mappings that connect your API Gateway APIs to the custom domain names
- Route 53 failover DNS records for the domain names
- A CloudWatch alarm in the secondary Region
- A Route 53 health check based on the CloudWatch alarm in the secondary Region

First, make sure that you have all of the required resources in the primary and secondary Regions. The secondary Region should contain the alarm and health check. This way, you don't depend on the primary Region to perform failover. For example AWS CloudFormation templates that create these resources, see `primary.yaml` and `secondary.yaml`.

⚠ **Important**

Before failover to the secondary Region, make sure that all required resources are available. Otherwise, your API won't be ready for traffic in the secondary Region.

Step 2: Initiate failover to the secondary Region

In the following example, the standby Region receives a CloudWatch metric and initiates failover. We use a custom metric that requires operator intervention to initiate failover.

```
aws cloudwatch put-metric-data \
    --metric-name Failover \
    --namespace HealthCheck \
    --unit Count \
    --value 1 \
    --region us-west-1
```

Replace the metric data with the corresponding data for the CloudWatch alarm you configured.

Step 3: Test the failover

Invoke your API and verify that you get a response from the secondary Region. If you used the example templates in step 1, the response changes from `{"message": "Hello from the primary Region!"}` to `{"message": "Hello from the secondary Region!"}` after failover.

```
curl https://my-api.example.com

{"message": "Hello from the secondary Region!"}
```

Step 4: Return to the primary region

To return to the primary Region, send a CloudWatch metric that causes the health check to pass.

```
aws cloudwatch put-metric-data \
    --metric-name Failover \
    --namespace HealthCheck \
    --unit Count \
    --value 0 \
    --region us-west-1
```

Replace the metric data with the corresponding data for the CloudWatch alarm you configured.

Invoke your API and verify that you get a response from the primary Region. If you used the example templates in step 1, the response changes from `{"message": "Hello from the secondary Region!"}` to `{"message": "Hello from the primary Region!"}`.

```
curl https://my-api.example.com

{"message": "Hello from the primary Region!"}
```

Next steps: Customize and test regularly

This example demonstrates one way to configure DNS failover. You can use a variety of CloudWatch metrics or HTTP endpoints for the health checks that manage failover. Regularly test your failover mechanisms to make sure that they work as expected, and that operators are familiar with your failover procedures.

Optimizing performance of REST APIs

After you've made your API available to be called, you might realize that it needs to be optimized to improve responsiveness. API Gateway provides a few strategies for optimizing your API, like response caching and payload compression. In this section, you can learn how to enable these capabilities.

Topics

Enabling API caching to enhance responsiveness

You can enable API caching in Amazon API Gateway to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API.

When you enable caching for a stage, API Gateway caches responses from your endpoint for a specified time-to-live (TTL) period, in seconds. API Gateway then responds to the request by looking up the endpoint response from the cache instead of making a request to your endpoint. The default TTL value for API caching is 300 seconds. The maximum TTL value is 3600 seconds. TTL=0 means caching is disabled.

> **Note**
>
> Caching is best-effort. You can use the `CacheHitCount` and `CacheMissCount` metrics in Amazon CloudWatch to monitor requests that API Gateway serves from the API cache.

The maximum size of a response that can be cached is 1048576 bytes. Cache data encryption may increase the size of the response when it is being cached.

This is a HIPAA Eligible Service. For more information about AWS, U.S. Health Insurance Portability and Accountability Act of 1996 (HIPAA), and using AWS services to process, store, and transmit protected health information (PHI), see HIPAA Overview.

Important

When you enable caching for a stage, only GET methods have caching enabled by default. This helps to ensure the safety and availability of your API. You can enable caching for other methods by overriding method settings.

Important

Caching is charged by the hour based on the cache size that you select. Caching is not eligible for the AWS Free Tier. For more information, see API Gateway Pricing.

Enable Amazon API Gateway caching

In API Gateway, you can enable caching for a specific stage.

When you enable caching, you must choose a cache capacity. In general, a larger capacity gives a better performance, but also costs more. For supported cache sizes, see cacheClusterSize in the *API Gateway API Reference*.

API Gateway enables caching by creating a dedicated cache instance. This process can take up to 4 minutes.

API Gateway changes caching capacity by removing the existing cache instance and creating a new one with a modified capacity. All existing cached data is deleted.

Note

The cache capacity affects the CPU, memory, and network bandwidth of the cache instance. As a result, the cache capacity can affect the performance of your cache.
API Gateway recommends that you run a 10-minute load test to verify that your cache capacity is appropriate for your workload. Ensure that traffic during the load test mirrors production traffic. For example, include ramp up, constant traffic, and traffic spikes. The load test should include responses that can be served from the cache, as well as unique responses that add items to the cache. Monitor the latency, 4xx, 5xx, cache hit, and cache miss metrics during the load test. Adjust your cache capacity as needed based on these

metrics. For more information about load testing, see How do I select the best API Gateway cache capacity to avoid hitting a rate limit?.

In the API Gateway console, you configure caching on the **Stages** page. You provision the stage cache and specify a default method-level cache setting. If you turn on the default method-level cache, method-level caching is turned on for all methods on your stage, unless that method has a method override. Any additional GET methods that you deploy to your stage will have a method-level cache. To configure method-level caching setting for specific methods of your stage, you can use method overrides. For more information about method overrides, see the section called “Override stage caching for method caching”.

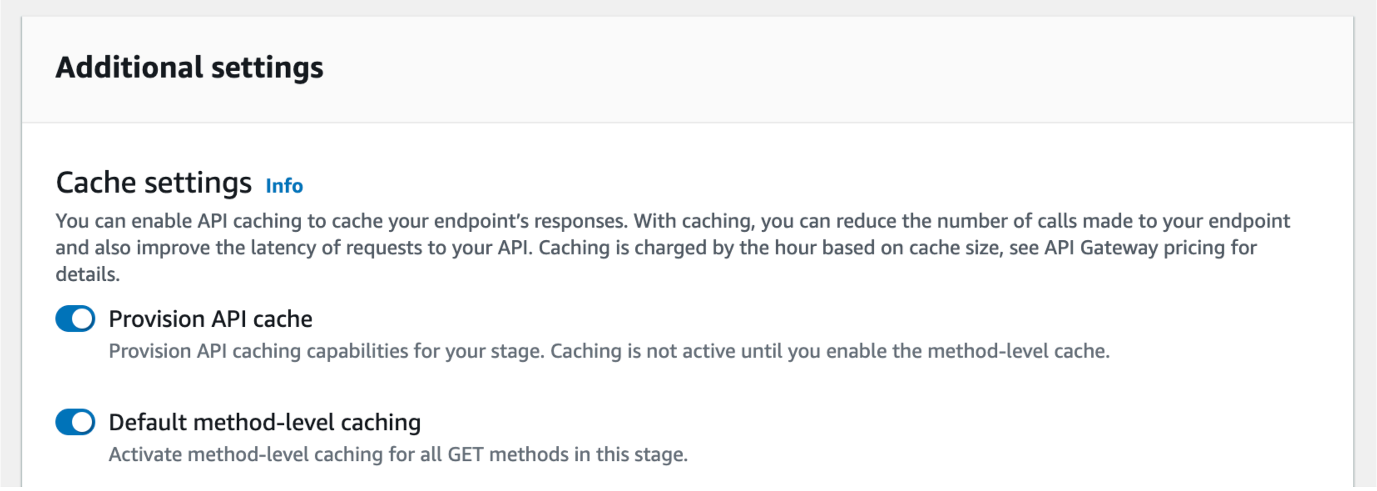
To configure API caching for a given stage:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Stages**.
3. In the **Stages** list for the API, choose the stage.
4. In the **Stage details** section, choose **Edit**.
5. Under **Additional settings**, for **Cache settings**, turn on **Provision API cache**.

   This provisions a cache cluster for your stage.
6. To activate caching for your stage, turn on **Default method-level caching**.

   This turns on method-level caching for all methods on your stage. Any additional GET methods that you deploy to this stage will have a method-level cache.

   > **Note**
   >
   > If you have an existing setting for a method-level cache, changing the default method-level caching setting doesn't affect that existing setting.



Additional settings

Cache settings Info

You can enable API caching to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API. Caching is charged by the hour based on cache size, see API Gateway pricing for details.

● Provision API cache
Provision API caching capabilities for your stage. Caching is not active until you enable the method-level cache.

● Default method-level caching
Activate method-level caching for all GET methods in this stage.

7. Choose **Save changes**.

> **Note**
>
> Creating or deleting a cache takes about 4 minutes for API Gateway to complete.
> When a cache is created, the **Cache cluster** value changes from `Create in progress` to `Active`. When cache deletion is completed, the **Cache cluster** value changes from `Delete in progress` to `Inactive`.
> When you turn on method-level caching for all methods on your stage, the **Default method-level caching** value changes to `Active`. If you turn off method-level caching for all methods on your stage, the **Default method-level caching** value changes to `Inactive`.
> If you have an existing setting for a method-level cache, changing the status of the cache doesn't affect that setting.

When you enable caching within a stage's **Cache settings**, only GET methods are cached. To ensure the safety and availability of your API, we recommend that you don't change this setting. However, you can enable caching for other methods by overriding method settings.

If you would like to verify if caching is functioning as expected, you have two general options:

- Inspect the CloudWatch metrics of **CacheHitCount** and **CacheMissCount** for your API and stage.
- Put a timestamp in the response.

> **Note**
>
> You should not use the X-Cache header from the CloudFront response to determine if your API is being served from your API Gateway cache instance.

Override API Gateway stage-level caching for method-level caching

You can override stage-level cache settings by turning on or turning off caching for a specific method. You can also modify the TTL period or turn encryption on or off for cached responses.

If you change the default method-level caching setting in the **Stage details**, it doesn't affect the method-level cache settings that have overrides.

If you anticipate that a method that you are caching will receive sensitive data in its responses, in **Cache Settings**, choose **Encrypt cache data**.

To configure API caching for individual methods using the console:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose the API.
3. Choose **Stages**.
4. In the **Stages** list for the API, expand the stage and choose a method in the API.
5. In the **Method overrides** section, choose **Edit**.
6. In the **Method settings** section, turn on or off **Enable method cache** or customize any other desired options.

   > **Note**
   >
   > Caching is not active until you provision a cache cluster for your stage.

7. Choose **Save**.

Use method or integration parameters as cache keys to index cached responses

When a cached method or integration has parameters, which can take the form of custom headers, URL paths, or query strings, you can use some or all of the parameters to form cache keys. API Gateway can cache the method's responses, depending on the parameter values used.

> **Note**
>
> Cache keys are required when setting up caching on a resource.

For example, suppose you have a request in the following format:

```
GET /users?type=... HTTP/1.1
host: example.com
...
```

In this request, `type` can take a value of `admin` or `regular`. If you include the `type` parameter as part of the cache key, the responses from `GET /users?type=admin` are cached separately from those from `GET /users?type=regular`.

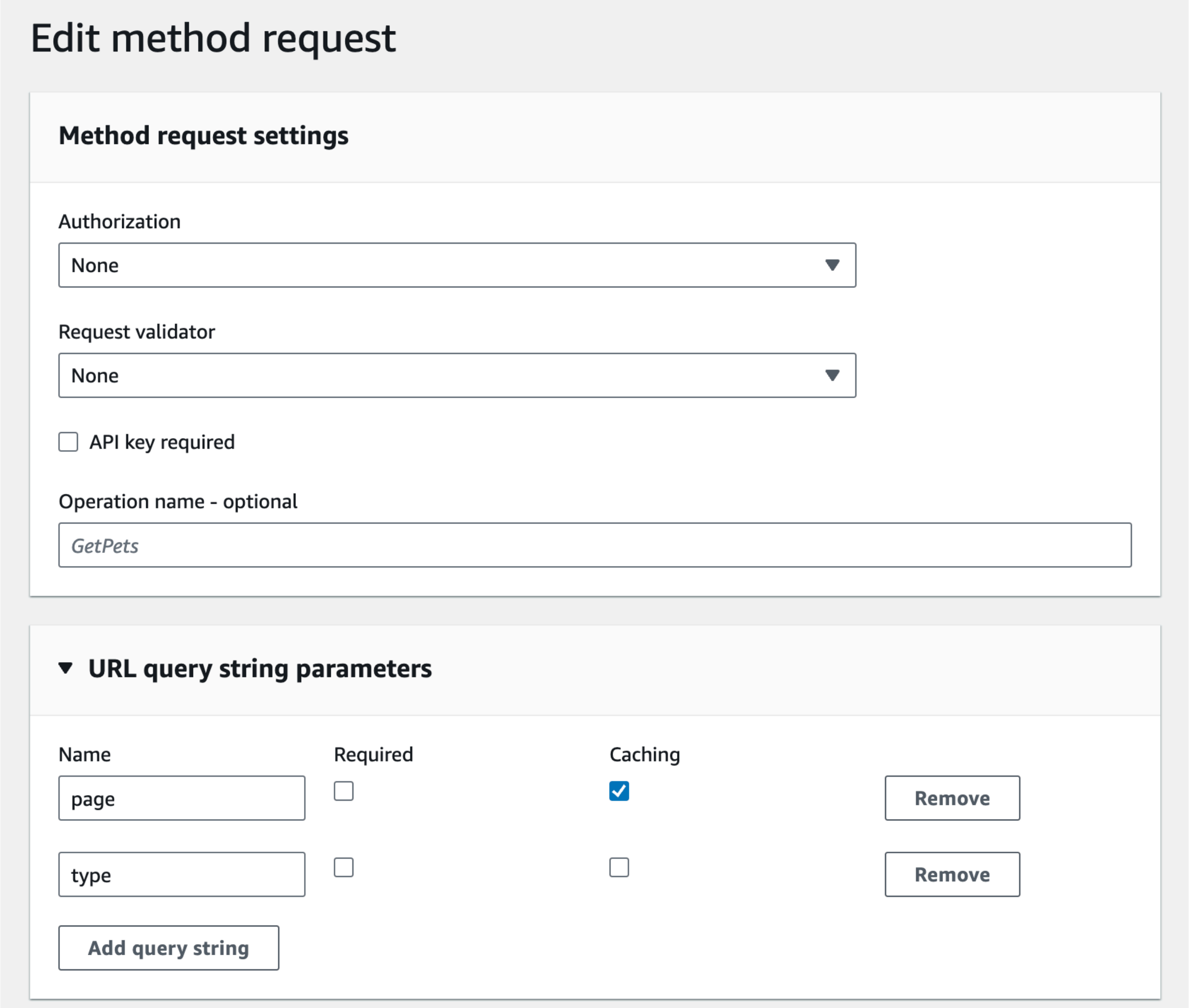
When a method or integration request takes more than one parameter, you can choose to include some or all of the parameters to create the cache key. For example, you can include only the `type` parameter in the cache key for the following request, made in the listed order within a TTL period:

```
GET /users?type=admin&department=A HTTP/1.1
host: example.com
...
```

The response from this request is cached and is used to serve the following request:

```
GET /users?type=admin&department=B HTTP/1.1
host: example.com
...
```

To include a method or integration request parameter as part of a cache key in the API Gateway console, select **Caching** after you add the parameter.



Edit method request

Method request settings

Authorization: None

Request validator: None

□ API key required

Operation name - optional: GetPets

URL query string parameters

Name	Required	Caching	
page	□	☑	Remove
type	□	□	Remove

Add query string

Flush the API stage cache in API Gateway

When API caching is enabled, you can flush your API stage's cache to ensure that your API's clients get the most recent responses from your integration endpoints.

To flush the API stage cache, choose the **Stage actions** menu, and then select **Flush stage cache**.

Note

After the cache is flushed, responses are serviced from the integration endpoint until the cache is built up again. During this period, the number of requests sent to the integration endpoint may increase. This may temporarily increase the overall latency of your API.

Invalidate an API Gateway cache entry

A client of your API can invalidate an existing cache entry and reload it from the integration endpoint for individual requests. The client must send a request that contains the `Cache-Control: max-age=0` header. The client receives the response directly from the integration endpoint instead of the cache, provided that the client is authorized to do so. This replaces the existing cache entry with the new response, which is fetched from the integration endpoint.

To grant permission for a client, attach a policy of the following format to an IAM execution role for the user.

Note

Cross-account cache invalidation is not supported.

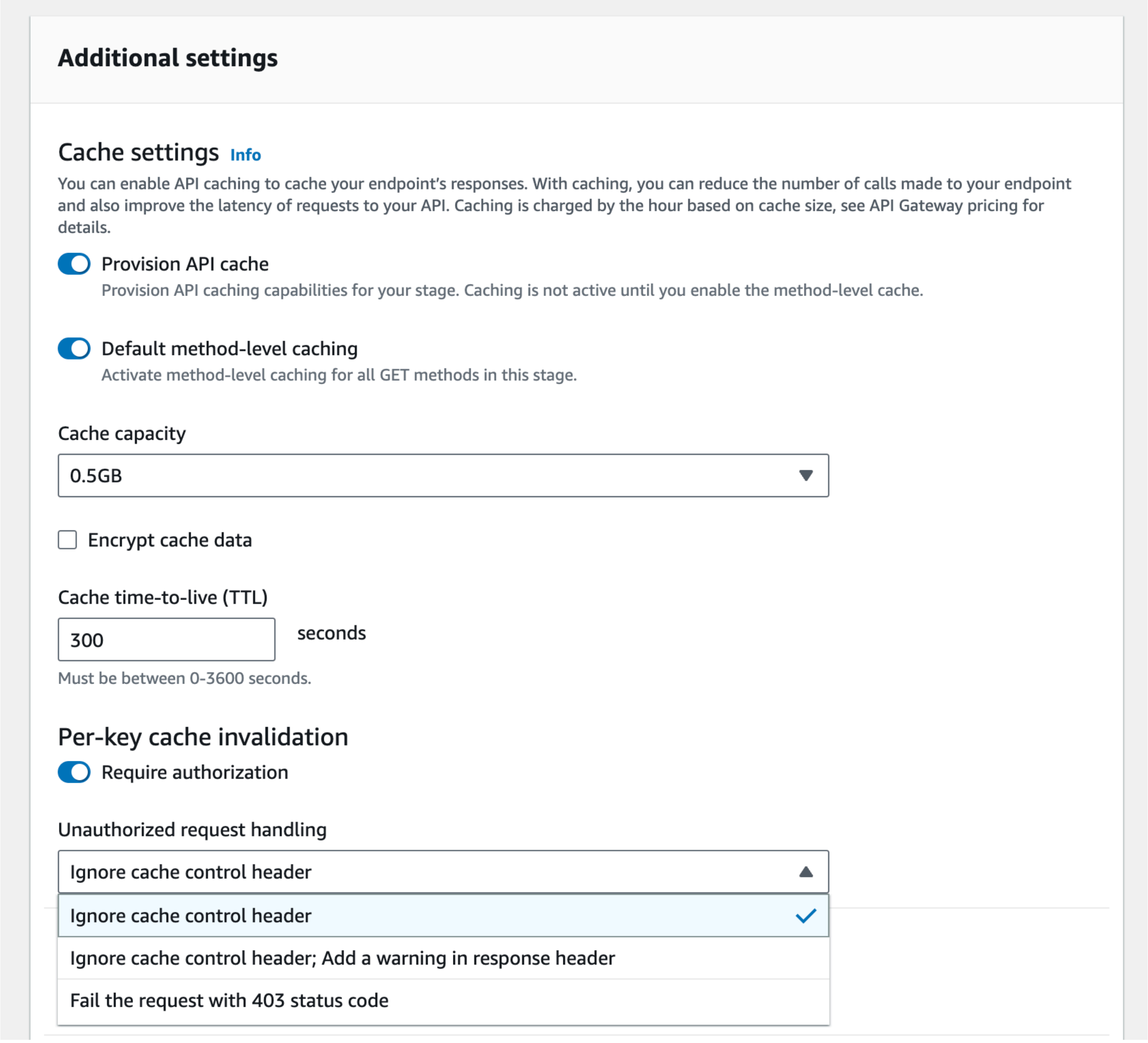
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:InvalidateCache"
      ],
      "Resource": [
        "arn:aws:execute-api:region:account-id:api-id/stage-name/GET/resource-path-specifier"
      ]
    }
  ]
}
```

This policy allows the API Gateway execution service to invalidate the cache for requests on the specified resource (or resources). To specify a group of targeted resources, use a wildcard (*) character for `account-id`, `api-id`, and other entries in the ARN value of `Resource`. For more information on how to set permissions for the API Gateway execution service, see Control access to an API with IAM permissions.

If you don't impose an `InvalidateCache` policy (or choose the **Require authorization** check box in the console), any client can invalidate the API cache. If most or all of the clients invalidate the API cache, this could significantly increase the latency of your API.

When the policy is in place, caching is enabled and authorization is required.

You can control how unauthorized requests are handled by choosing an option from **Unauthorized request handling** in the API Gateway console.



Additional settings

Cache settings Info

You can enable API caching to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API. Caching is charged by the hour based on cache size, see API Gateway pricing for details.

● Provision API cache

Provision API caching capabilities for your stage. Caching is not active until you enable the method-level cache.

● Default method-level caching

Activate method-level caching for all GET methods in this stage.

Cache capacity

0.5GB

□ Encrypt cache data

Cache time-to-live (TTL)

300 seconds

Must be between 0-3600 seconds.

Per-key cache invalidation

● Require authorization

Unauthorized request handling

Ignore cache control header

- Ignore cache control header ✓
- Ignore cache control header; Add a warning in response header
- Fail the request with 403 status code

The three options result in the following behaviors:

- **Fail the request with 403 status code**: returns a 403 Unauthorized response.

  To set this option using the API, use `FAIL_WITH_403`.

- **Ignore cache control header; Add a warning in response header**: process the request and add a warning header in the response.

  To set this option using the API, use `SUCCEED_WITH_RESPONSE_HEADER`.

- **Ignore cache control header**: process the request and do not add a warning header in the response.

  To set this option using the API, use SUCCEED_WITHOUT_RESPONSE_HEADER.

Enabling payload compression for an API

API Gateway allows your client to call your API with compressed payloads by using one of the supported content codings. By default, API Gateway supports decompression of the method request payload. However, you must configure your API to enable compression of the method response payload.

To enable compression on an API, set the minimumCompressionsSize property to a non-negative integer between 0 and 10485760 (10M bytes) when you create the API or after you've created the API. To disable compression on the API, set the minimumCompressionSize to null or remove it altogether. You can enable or disable compression for an API by using the API Gateway console, the AWS CLI, or the API Gateway REST API.

If you want the compression applied on a payload of any size, set the minimumCompressionSize value to zero. However, compressing data of a small size might actually increase the final data size. Furthermore, compression in API Gateway and decompression in the client might increase overall latency and require more computing times. You should run test cases against your API to determine an optimal value.

The client can submit an API request with a compressed payload and an appropriate Content-Encoding header for API Gateway to decompress and apply applicable mapping templates, before passing the request to the integration endpoint. After the compression is enabled and the API is deployed, the client can receive an API response with a compressed payload if it specifies an appropriate Accept-Encoding header in the method request.

When the integration endpoint expects and returns uncompressed JSON payloads, any mapping template that's configured for an uncompressed JSON payload is applicable to the compressed payload. For a compressed method request payload, API Gateway decompresses the payload, applies the mapping template, and passes the mapped request to the integration endpoint. For an uncompressed integration response payload, API Gateway applies the mapping template, compresses the mapped payload, and returns the compressed payload to the client.

Topics

- Enable payload compression for an API
- Call an API method with a compressed payload
- Receive an API response with a compressed payload

Enable payload compression for an API

You can enable compression for an API using the API Gateway console, the AWS CLI, or an AWS SDK.

For an existing API, you must deploy the API after enabling the compression in order for the change to take effect. For a new API, you can deploy the API after the API setup is complete.

> **Note**
>
> The highest-priority content encoding must be one supported by API Gateway. If it is not, compression is not applied to the response payload.

Topics

- Enable payload compression for an API using the API Gateway console
- Enable payload compression for an API using AWS CLI
- Content codings supported by API Gateway

Enable payload compression for an API using the API Gateway console

The following procedure describes how to enable payload compression for an API.

To enable payload compression by using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an existing API or create a new one.
3. In the main navigation pane, choose **API settings**.
4. In the **API details** section, choose **Edit**.
5. Turn on **Content encoding** to enable payload compression. For **Minimum body size**, enter a number for the minimum compression size (in bytes). To turn off compression, turn off the **Content encoding** option.

6. Choose **Save changes**.

Enable payload compression for an API using AWS CLI

To use the AWS CLI to create a new API and enable compression, call the `create-rest-api` command as follows:

```
aws apigateway create-rest-api \
    --name "My test API" \
    --minimum-compression-size 0
```

To use the AWS CLI to enable compression on an existing API, call the `update-rest-api` command as follows:

```
aws apigateway update-rest-api \
    --rest-api-id 1234567890 \
    --patch-operations op=replace,path=/minimumCompressionSize,value=0
```

The `minimumCompressionSize` property has a non-negative integer value between 0 and 10485760 (10M bytes). It measures the compression threshold. If the payload size is smaller than this value, compression or decompression are not applied on the payload. Setting it to zero allows compression for any payload size.

To use the AWS CLI to disable compression, call the `update-rest-api` command as follows:

```
aws apigateway update-rest-api \
    --rest-api-id 1234567890 \
    --patch-operations op=replace,path=/minimumCompressionSize,value=
```

You can also set `value` to an empty string `""` or omit the `value` property altogether in the preceding call.

Content codings supported by API Gateway

API Gateway supports the following content codings:

- `deflate`
- `gzip`
- `identity`

API Gateway also supports the following `Accept-Encoding` header format, according to the [RFC 7231](#) specification:

- `Accept-Encoding:deflate,gzip`
- `Accept-Encoding:`
- `Accept-Encoding:*`
- `Accept-Encoding:deflate;q=0.5,gzip;q=1.0`
- `Accept-Encoding:gzip;q=1.0,identity;q=0.5,*;q=0`

Call an API method with a compressed payload

To make an API request with a compressed payload, the client must set the `Content-Encoding` header with one of the [supported content codings](#).

Suppose that you're an API client and want to call the PetStore API method (`POST /pets`). Don't call the method by using the following JSON output:

```
POST /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Content-Length: ...

{
  "type": "dog",
  "price": 249.99
}
```

Instead, you can call the method with the same payload compressed by using the GZIP coding:

```
POST /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Content-Encoding:gzip
Content-Length: ...

���RPP*�,HU�RPJ�OW��e&���L,�,-y�j
```

When API Gateway receives the request, it verifies if the specified content coding is supported. Then, it attempts to decompress the payload with the specified content coding. If the decompression is successful, it dispatches the request to the integration endpoint. If the specified coding isn't supported or the supplied payload isn't compressed with specified coding, API Gateway returns the `415 Unsupported Media Type` error response. The error is not logged to CloudWatch Logs, if it occurs in the early phase of decompression before your API and stage are identified.

Receive an API response with a compressed payload

When making a request on a compression-enabled API, the client can choose to receive a compressed response payload of a specific format by specifying an `Accept-Encoding` header with a supported content coding.

API Gateway only compresses the response payload when the following conditions are satisfied:

- The incoming request has the `Accept-Encoding` header with a supported content coding and format.

> **Note**
>
> If the header is not set, the default value is `*` as defined in RFC 7231. In such a case, API Gateway does not compress the payload. Some browser or client may add `Accept-Encoding` (for example, `Accept-Encoding:gzip, deflate, br`) automatically to compression-enabled requests. This can trigger the payload compression in API Gateway. Without an explicit specification of supported `Accept-Encoding` header values, API Gateway does not compress the payload.

- The `minimumCompressionSize` is set on the API to enable compression.
- The integration response doesn't have a `Content-Encoding` header.
- The size of an integration response payload, after the applicable mapping template is applied, is greater than or equal to the specified `minimumCompressionSize` value.

API Gateway applies any mapping template that's configured for the integration response before compressing the payload. If the integration response contains a `Content-Encoding` header, API Gateway assumes that the integration response payload is already compressed and skips the compression processing.

An example is the PetStore API example and the following request:

```
GET /pets
```

```
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Accept: application/json
```

The backend responds to the request with an uncompressed JSON payload that's similar to the following:

```
200 OK

[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

To receive this output as a compressed payload, your API client can submit a request as follows:

```
GET /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Accept-Encoding:gzip
```

The client receives the response with a `Content-Encoding` header and GZIP-encoded payload that are similar to the following:

```
200 OK
Content-Encoding:gzip
...

���RP�

J�)JV
```

```
�:P^IeA*������+(�L �X�YZ�ku0L0B7!9��C#�&����Y��a���^�X
```

When the response payload is compressed, only the compressed data size is billed for data transfer.

Distributing your REST API to clients

This section provides details about distributing your API Gateway APIs to your customers. Distributing your API includes generating SDKs for your customers to download and integrate with their client applications, documenting your API so customers know how to call it from their client applications, and making your API available as part of product offerings.

Topics

- Creating and using usage plans with API keys
- Documenting REST APIs
- Generating an SDK for a REST API in API Gateway
- Sell your API Gateway APIs through AWS Marketplace

Creating and using usage plans with API keys

After you create, test, and deploy your APIs, you can use API Gateway usage plans to make them available as product offerings for your customers. You can configure usage plans and API keys to allow customers to access selected APIs, and begin throttling requests to those APIs based on defined limits and quotas. These can be set at the API, or API method level.

What are usage plans and API keys?

A *usage plan* specifies who can access one or more deployed API stages and methods—and optionally sets the target request rate to start throttling requests. The plan uses API keys to identify API clients and who can access the associated API stages for each key.

API keys are alphanumeric string values that you distribute to application developer customers to grant access to your API. You can use API keys together with Lambda authorizers, IAM roles, or Amazon Cognito to control access to your APIs. API Gateway can generate API keys on your behalf, or you can import them from a CSV file. You can generate an API key in API Gateway, or import it into API Gateway from an external source. For more information, see the section called “Set up API keys using the API Gateway console”.

An API key has a name and a value. (The terms "API key" and "API key value" are often used interchangeably.) The name cannot exceed 1024 characters. The value is an alphanumeric string between 20 and 128 characters, for example, `apikey1234abcdefghij0123456789`.

> ⚠ **Important**
>
> API key values must be unique. If you try to create two API keys with different names and the same value, API Gateway considers them to be the same API key.
> An API key can be associated with more than one usage plan. A usage plan can be associated with more than one stage. However, a given API key can only be associated with one usage plan for each stage of your API.

A *throttling limit* sets the target point at which request throttling should start. This can be set at the API or API method level.

A *quota limit* sets the target maximum number of requests with a given API key that can be submitted within a specified time interval. You can configure individual API methods to require API key authorization based on usage plan configuration.

Throttling and quota limits apply to requests for individual API keys that are aggregated across all API stages within a usage plan.

> ⓘ **Note**
>
> Usage plan throttling and quotas are not hard limits, and are applied on a best-effort basis. In some cases, clients can exceed the quotas that you set. Don’t rely on usage plan quotas or throttling to control costs or block access to an API. Consider using AWS Budgets to monitor costs and AWS WAF to manage API requests.

Best practices for API keys and usage plans

The following are suggested best practices to follow when using API keys and usage plans.

> ⚠ **Important**
>
> - Don't use API keys for authentication or authorization to control access to your APIs. If you have multiple APIs in a usage plan, a user with a valid API key for one API in that

usage plan can access *all* APIs in that usage plan. Instead, to control access to your API, use an IAM role, a Lambda authorizer, or an Amazon Cognito user pool.

- Use API keys that API Gateway generates. API keys shouldn't include confidential information; clients typically transmit them in headers that can be logged.

- If you're using a developer portal to publish your APIs, note that all APIs in a given usage plan are subscribable, even if you haven't made them visible to your customers.
- In some cases, clients can exceed the quotas that you set. Don’t rely on usage plans to control costs. Consider using AWS Budgets to monitor costs and AWS WAF to manage API requests.
- After you add an API key to a usage plan, the update operation might take a few minutes to complete.

Steps to configure a usage plan

The following steps outline how you, as the API owner, create and configure a usage plan for your customers.

To configure a usage plan

1. Create one or more APIs, configure the methods to require an API key, and deploy the APIs to stages.
2. Generate or import API keys to distribute to application developers (your customers) who will be using your API.
3. Create the usage plan with the desired throttle and quota limits.
4. Associate API stages and API keys with the usage plan.

Callers of the API must supply an assigned API key in the `x-api-key` header in requests to the API.

Note

To include API methods in a usage plan, you must configure individual API methods to require an API key. For best practices to consider, see the section called “Best practices for API keys and usage plans”.

Choose an API key source

When you associate a usage plan with an API and enable API keys on API methods, every incoming request to the API must contain an API key. API Gateway reads the key and compares it against the keys in the usage plan. If there is a match, API Gateway throttles the requests based on the plan's request limit and quota. Otherwise, it throws an `InvalidKeyParameter` exception. As a result, the caller receives a `403 Forbidden` response.

Your API Gateway API can receive API keys from one of two sources:

`HEADER`

You distribute API keys to your customers and require them to pass the API key as the `X-API-Key` header of each incoming request.

`AUTHORIZER`

You have a Lambda authorizer return the API key as part of the authorization response. For more information on the authorization response, see the section called “Output from an Amazon API Gateway Lambda authorizer”.

> **Note**
>
> For best practices to consider, see the section called “Best practices for API keys and usage plans”.

To choose an API key source for an API by using the API Gateway console

1. Sign in to the API Gateway console.
2. Choose an existing API or create a new one.
3. In the main navigation pane, choose **API settings**.
4. In the **API details** section, choose **Edit**.
5. Under **API key source**, select `Header` or `Authorizer` from the dropdown list.
6. Choose **Save changes**.

To choose an API key source for an API by using the AWS CLI, call the `update-rest-api` command as follows:

```
aws apigateway update-rest-api --rest-api-id 1234123412 --patch-operations
 op=replace,path=/apiKeySource,value=AUTHORIZER
```

To have the client submit an API key, set the `value` to `HEADER` in the preceding CLI command.

To choose an API key source for an API by using the API Gateway REST API, call `restapi:update` as follows:

```
PATCH /restapis/fugvjdxtri/ HTTP/1.1
Content-Type: application/json
Host: apigateway.us-east-1.amazonaws.com
X-Amz-Date: 20160603T205348Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20160603/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature={sig4_hash}

{
  "patchOperations" : [
    {
        "op" : "replace",
        "path" : "/apiKeySource",
        "value" : "HEADER"
    }
  ]
}
```

To have an authorizer return an API key, set the `value` to `AUTHORIZER` in the previous `patchOperations` input.

Depending on the API key source type you choose, use one of the following procedures to use header-sourced API keys or authorizer-returned API keys in method invocation:

To use header-sourced API keys:

1. Create an API with desired API methods, and then deploy the API to a stage.
2. Create a new usage plan or choose an existing one. Add the deployed API stage to the usage plan. Attach an API key to the usage plan or choose an existing API key in the plan. Note the chosen API key value.
3. Set up API methods to require an API key.

4. Redeploy the API to the same stage. If you deploy the API to a new stage, make sure to update the usage plan to attach the new API stage.

The client can now call the API methods while supplying the `x-api-key` header with the chosen API key as the header value.

To use authorizer-sourced API keys:

1. Create an API with desired API methods, and then deploy the API to a stage.
2. Create a new usage plan or choose an existing one. Add the deployed API stage to the usage plan. Attach an API key to the usage plan or choose an existing API key in the plan. Note the chosen API key value.
3. Create a token-based Lambda authorizer. Include, `usageIdentifierKey:`*`{api-key}`* as a root-level property of the authorization response. For instructions on creating a token-based authorizer, see the section called "EXAMPLE: Create a token-based Lambda authorizer function".
4. Set up API methods to require an API key and enable the Lambda authorizer on the methods as well.
5. Redeploy the API to the same stage. If you deploy the API to a new stage, make sure to update the usage plan to attach the new API stage.

The client can now call the API key-required methods without explicitly supplying any API key. The authorizer-returned API key is used automatically.

Set up API keys using the API Gateway console

To set up API keys, do the following:

- Configure API methods to require an API key.
- Create or import an API key for the API in a region.

Before setting up API keys, you must have created an API and deployed it to a stage. After you create an API key value, it cannot be changed.

For instructions on how to create and deploy an API by using the API Gateway console, see Creating a REST API in Amazon API Gateway and Deploying a REST API in Amazon API Gateway, respectively.

After you create an API key, you must associate it with a usage plan. For more information, see Create, configure, and test usage plans with the API Gateway console.

> ⓘ **Note**
>
> For best practices to consider, see the section called “Best practices for API keys and usage plans”.

Topics

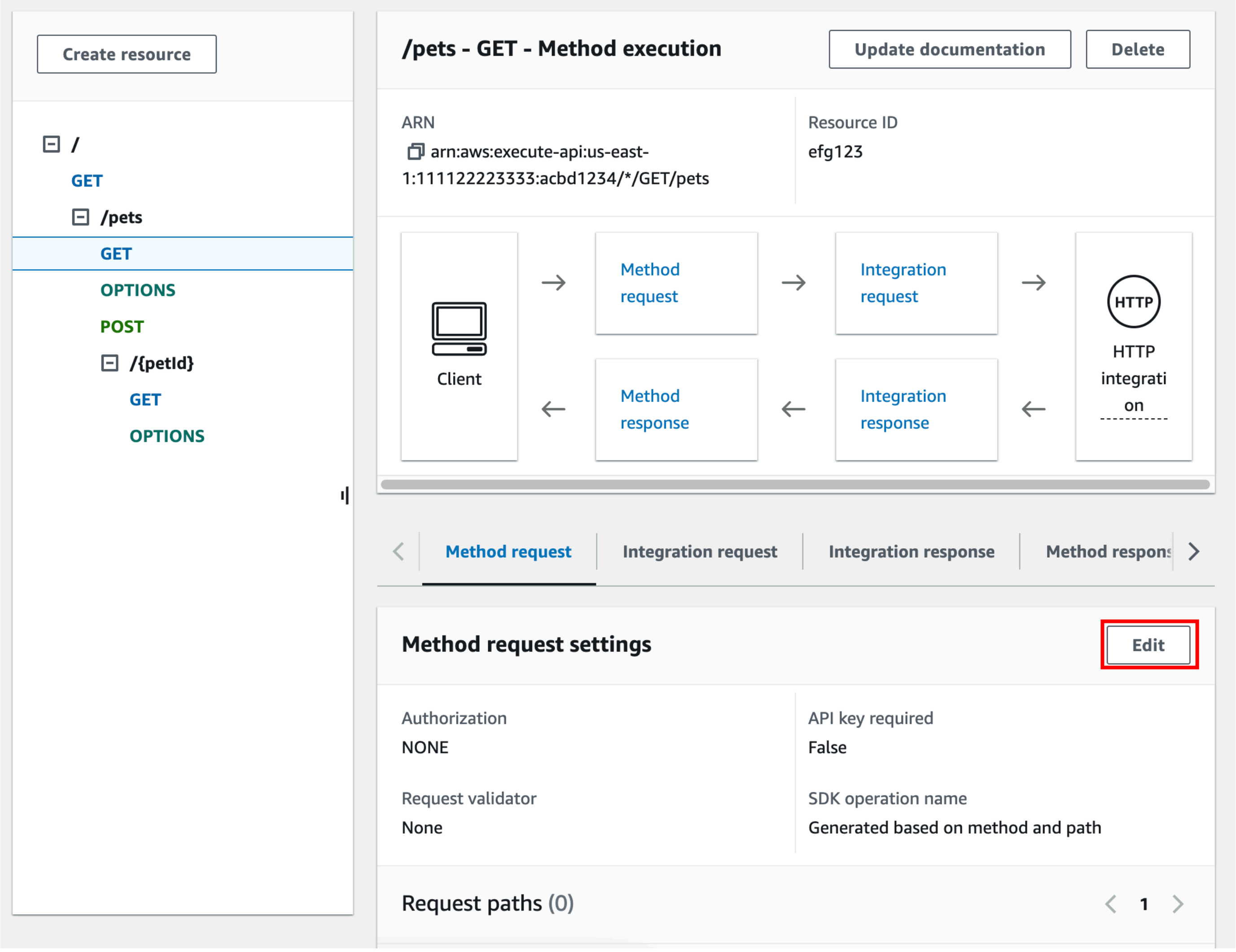
- Require API key on a method
- Create an API key
- Import API keys

Require API key on a method

The following procedure describes how to configure an API method to require an API key.

To configure an API method to require an API key

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the API Gateway main navigation pane, choose **Resources**.
4. Under **Resources**, create a new method or choose an existing one.
5. On the **Method request** tab, under **Method request settings**, choose **Edit**.



/pets - GET - Method execution

ARN
arn:aws:execute-api:us-east-1:111122223333:acbd1234/*/GET/pets

Resource ID
efg123

Method request settings

Authorization
NONE

API key required
False

Request validator
None

SDK operation name
Generated based on method and path

Request paths (0)

6. Select **API key required**.
7. Choose **Save**.
8. Deploy or redeploy the API for the requirement to take effect.

If the **API key required** option is set to `false` and you don't execute the previous steps, any API key that's associated with an API stage isn't used for the method.

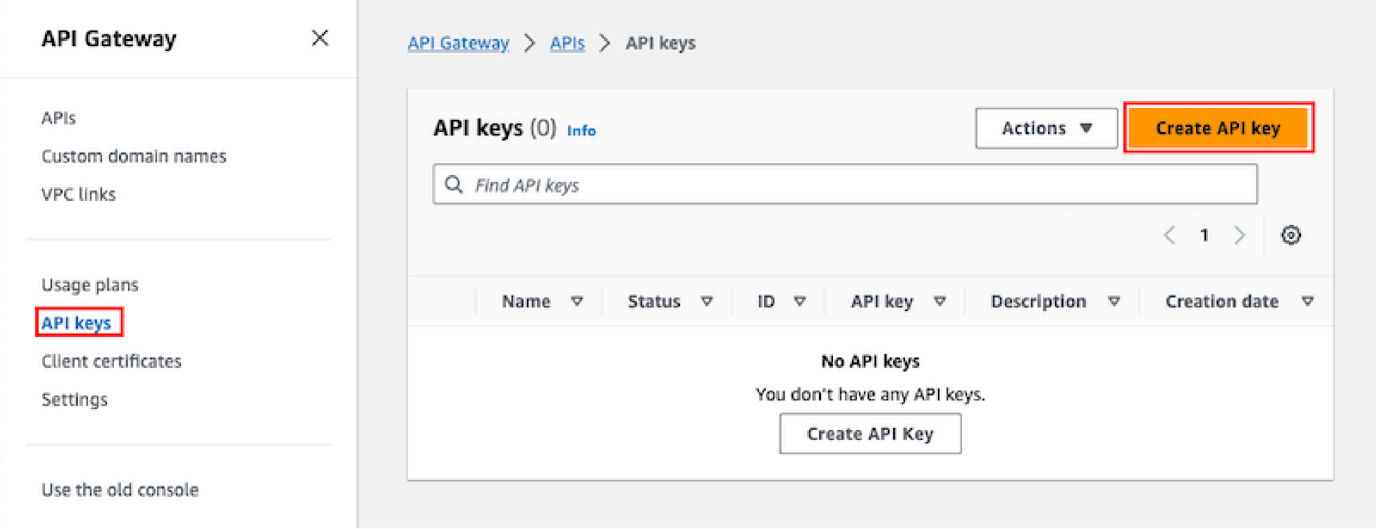
Create an API key

If you've already created or imported API keys for use with usage plans, you can skip this and the next procedure.

To create an API key

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.

2. Choose a REST API.
3. In the API Gateway main navigation pane, choose **API keys**.
4. Choose **Create API key**.



No API keys

You don't have any API keys.

5. For **Name**, enter a name.
6. (Optional) For **Description**, enter a description.
7. For **API key**, choose **Auto generate** to have API Gateway generate the key value, or choose **Custom** to create your own key value.
8. Choose **Save**.

Import API keys

The following procedure describes how to import API keys to use with usage plans.

To import API keys

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. In the main navigation pane, choose **API keys**.
4. Choose the **Actions** dropdown menu, and then choose **Import API keys**.
5. To load a comma-separated key file, choose **Choose file**. You can also enter the keys in the text editor. For information about the file format, see the section called "API Gateway API key file format".
6. Choose **Fail on warnings** to stop the import when there's an error, or choose **Ignore warnings** to continue to import valid key entries when there's an warning.

7. Choose **Import** to import your API keys.

Create, configure, and test usage plans with the API Gateway console

Before creating a usage plan, make sure that you've set up the desired API keys. For more information, see Set up API keys using the API Gateway console.

This section describes how to create and use a usage plan by using the API Gateway console.

Topics

Migrate your API to default usage plans (if needed)

If you started to use API Gateway *after* the usage plans feature was rolled out on August 11, 2016, you will automatically have usage plans enabled for you in all supported Regions.

If you started to use API Gateway before that date, you might need to migrate to default usage plans. You'll be prompted with the **Enable Usage Plans** option before using usage plans for the first time in the selected Region. When you enable this option, you have default usage plans created for every unique API stage that's associated with existing API keys. In the default usage plan, no throttle or quota limits are set initially, and the associations between the API keys and API stages are copied to the usage plans. The API behaves the same as before. However, you must use the `UsagePlan` `apiStages` property to associate specified API stage values (`apiId` and `stage`) with included API keys (via `UsagePlanKey`), instead of using the ApiKey `stageKeys` property.

To check whether you've already migrated to default usage plans, use the `get-account` CLI command. In the command output, the `features` list includes an entry of `"UsagePlans"` when usage plans are enabled.

You can also migrate your APIs to default usage plans by using the AWS CLI as follows:

To migrate to default usage plans using the AWS CLI

1. Call this CLI command: `update-account`.

2. For the `cli-input-json` parameter, use the following JSON:

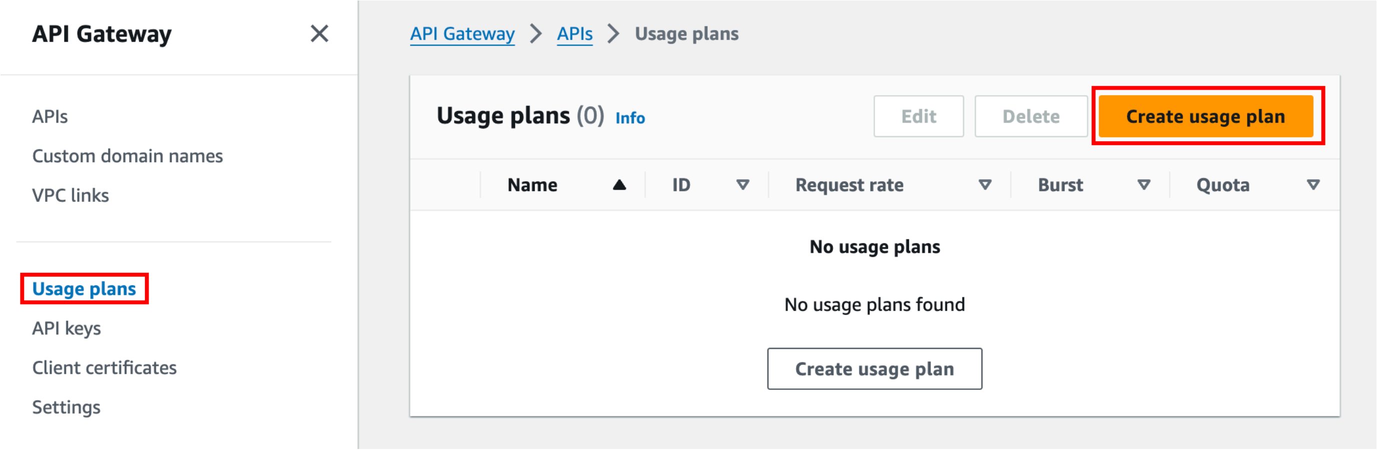
```
[
    {
        "op": "add",
        "path": "/features",
        "value": "UsagePlans"
    }
]
```

Create a usage plan

The following procedure describes how to create a usage plan.

To create a usage plan

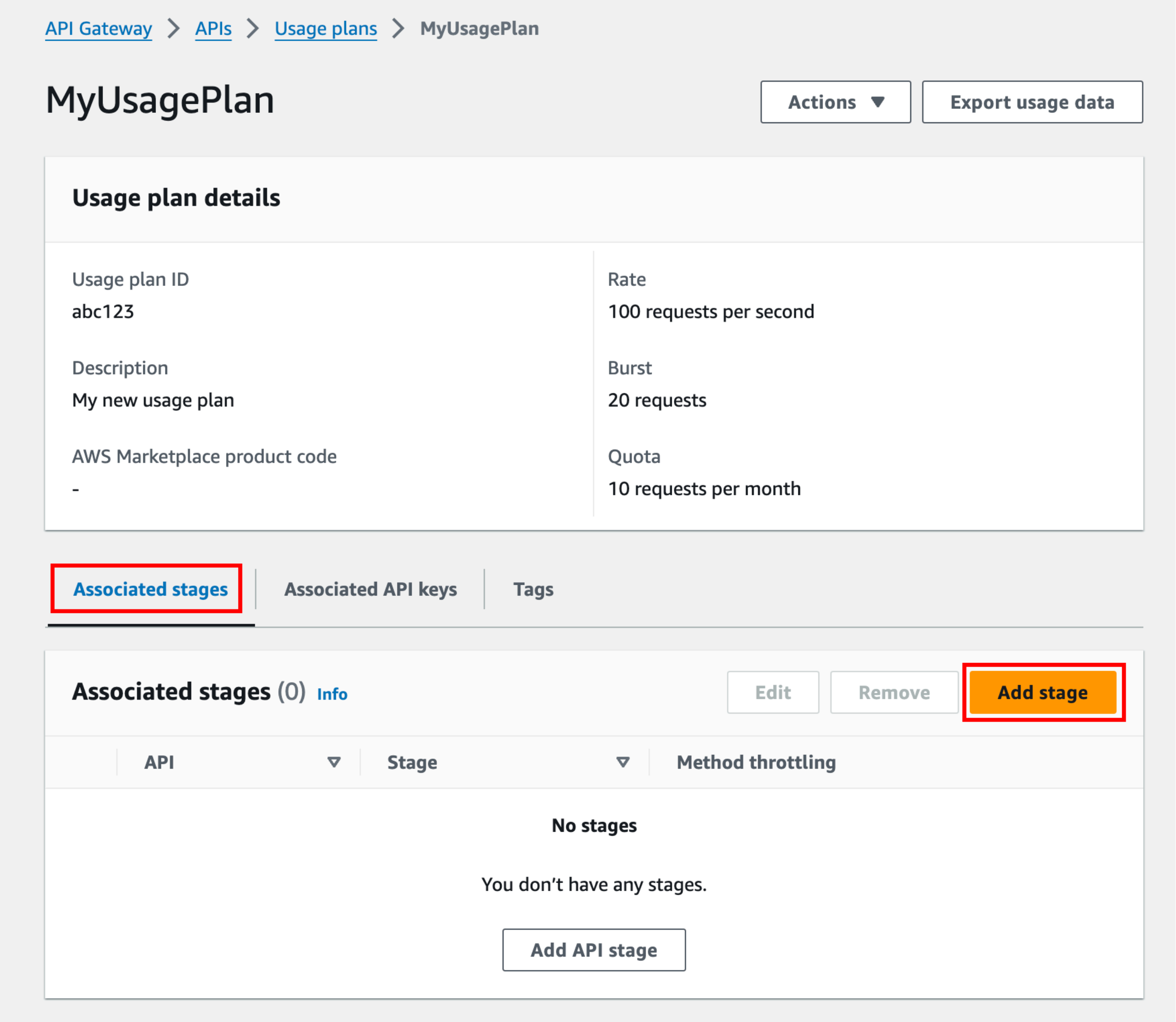
1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the API Gateway main navigation pane, choose **Usage plans**, and then choose **Create usage plan**.



3. For **Name**, enter a name.
4. (Optional) For **Description**, enter a description.
5. By default, usage plans enable throttling. Enter a **Rate** and a **Burst** for your usage plan. Choose **Throttling** to turn off throttling.
6. By default, usage plans enable a quota for a time period. For **Requests**, enter the total number of requests that a user can make in the time period of your usage plan. Choose **Quota** to turn off the quota.
7. Choose **Create usage plan**.

To add a stage to the usage plan

1. Select your usage plan.
2. Under the **Associated stages** tab, choose **Add stage**.



Usage plan details

Usage plan ID abc123	Rate 100 requests per second
Description My new usage plan	Burst 20 requests
AWS Marketplace product code -	Quota 10 requests per month

Associated stages (0) Info

API	Stage	Method throttling

No stages

You don't have any stages.

3. For **API**, select an API.
4. For **Stage**, select a stage.
5. (Optional) To turn on method-level throttling, do the following:

   a. Choose **Method-level throttling**, and then choose **Add method**.

   b. For **Resource**, select a resource from your API.

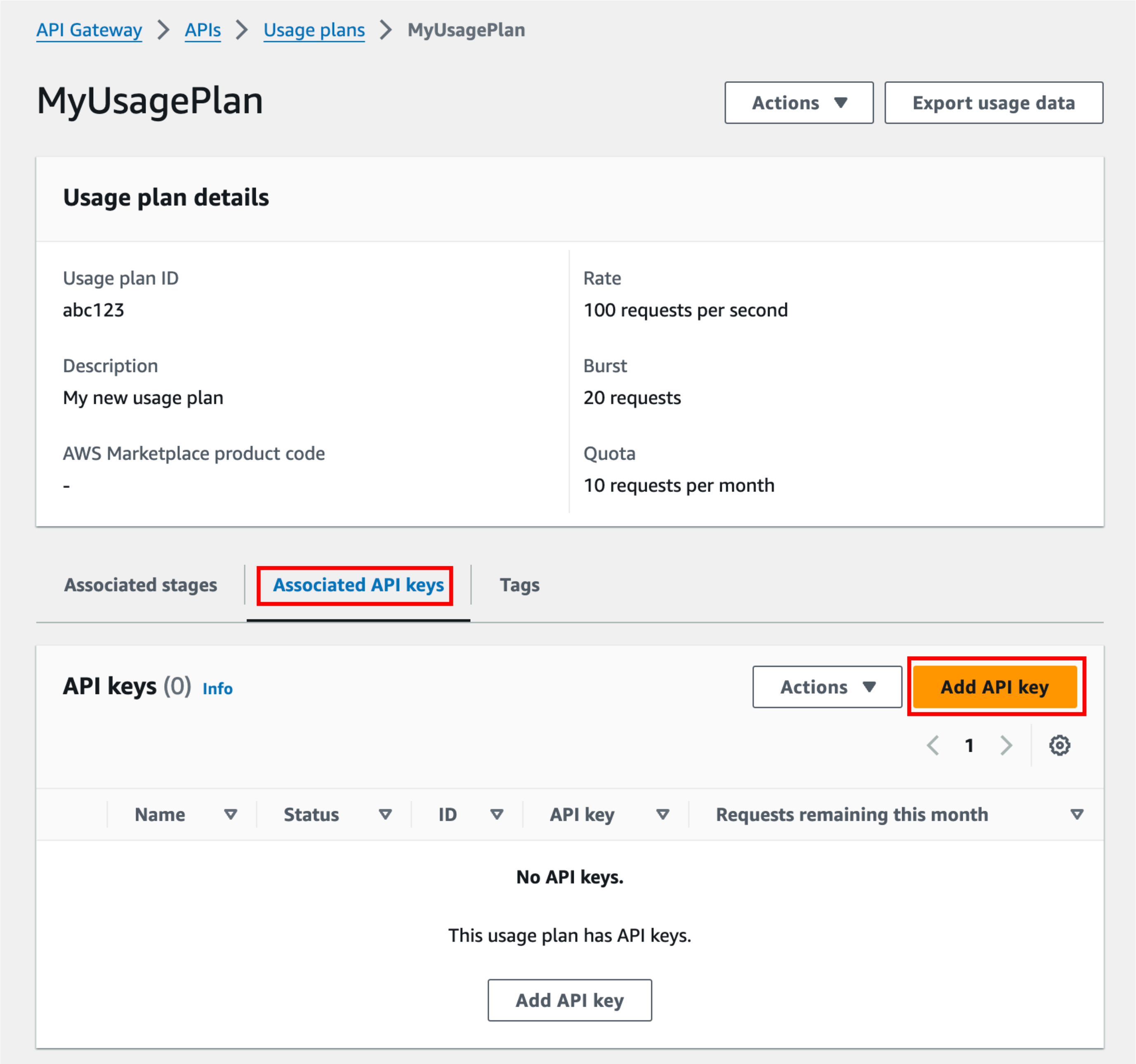
   c. For **Method**, select a method from your API.

d. Enter a **Rate** and a **Burst** for your usage plan.

6. Choose **Add to usage plan**.

To add a key to the usage plan

1. Under the **Associated API keys** tab, choose **Add API key**.



MyUsagePlan

Usage plan details

Usage plan ID
abc123

Description
My new usage plan

AWS Marketplace product code
-

Rate
100 requests per second

Burst
20 requests

Quota
10 requests per month

API keys (0)

Name	Status	ID	API key	Requests remaining this month

No API keys.

This usage plan has API keys.

2. a. To associate an existing key to your usage plan, select **Add existing key**, and then select your existing key from the dropdown menu.

   b. To create a new API key, select **Create and add new key**, and then create a new key. For more information on how to create a new key, see Create an API key.

3. Choose **Add API key**.

Test a usage plan

To test the usage plan, you can use an AWS SDK, AWS CLI, or a REST API client like Postman. For an example of using Postman to test the usage plan, see Test usage plans.

Maintain a usage plan

Maintaining a usage plan involves monitoring the used and remaining quotas over a given time period and, if needed, extending the remaining quotas by a specified amount. The following procedures describe how to monitor quotas.

To monitor used and remaining quotas

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the API Gateway main navigation pane, choose **Usage plans**.
3. Select a usage plan.
4. Choose the **Associated API keys** tab to see the number of request remaining for the time period for each key.
5. (Optional) Choose **Export usage data**, and then choose a **From** date and a **To** date. Then choose **JSON** or **CSV** for the exported data format, and then choose **Export**.

   The following example shows an exported file.

```
{
  "thisPeriod": {
    "px1KW6...qBazOJH": [
      [
        0,
        5000
      ],
      [
        0,
        5000
      ],
      [
        0,
        10
      ]
    ]
  },
  "startDate": "2016-08-01",
  "endDate": "2016-08-03"
}
```

The usage data in the example shows the daily usage data for an API client, as identified by the API key (`px1KW6...qBazOJH`), between August 1, 2016 and August 3, 2016. Each daily usage data shows used and remaining quotas. In this example, the subscriber hasn't used any allotted quotas yet, and the API owner or administrator has reduced the remaining quota from 5000 to 10 on the third day.

The following procedures describe how to modify quotas.

To extend the remaining quotas

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. In the API Gateway main navigation pane, choose **Usage plans**.
3. Select a usage plan.
4. Choose the **Associated API keys** tab to see the number of request remaining for the time period for each key.
5. Select an API key, and then choose **Grant usage extension**.
6. Enter a number for the **Remaining requests** quota. You can increase the renaming requests or decrease the remaining requests for the time period of your usage plan.
7. Choose **Update quota**.

Set up API keys using the API Gateway REST API

To set up API keys, do the following:

- Configure API methods to require an API key.
- Create or import an API key for the API in a region.

Before setting up API keys, you must have created an API and deployed it to a stage. After you create an API key value, it cannot be changed.

For the REST API calls to create and deploy an API, see `restapi:create` and `deployment:create`, respectively.

> **Note**
>
> For best practices to consider, see the section called "Best practices for API keys and usage plans".

Topics

- Require an API key on a method
- Create or import API keys

Require an API key on a method

To require an API key on a method, do one of the following:

- Call `method:put` to create a method. Set `apiKeyRequired` to `true` in the request payload.
- Call `method:update` to set `apiKeyRequired` to `true`.

Create or import API keys

To create or import an API key, do one of the following:

- Call `apikey:create` to create an API key.
- Call `apikey:import` to import an API key from a file. For the file format, see API Gateway API key file format.

You cannot change the value of the new API key. To learn how to configure a usage plan, see Create, configure, and test usage plans using the API Gateway CLI and REST API.

Create, configure, and test usage plans using the API Gateway CLI and REST API

Before configuring a usage plan, you must have already done the following: set up methods of a selected API to require API keys, deployed or redeployed the API to a stage, and created or imported one or more API keys. For more information, see Set up API keys using the API Gateway REST API.

To configure a usage plan by using the API Gateway REST API, use the following instructions, assuming that you've already created the APIs to be added to the usage plan.

Topics

- Migrate to default usage plans
- Create a usage plan
- Manage a usage plan by using the AWS CLI
- Test usage plans

Migrate to default usage plans

When creating a usage plan the first time, you can migrate existing API stages that are associated with selected API keys to a usage plan by calling `account:update` with the following body:

```
{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/features",
    "value" : "UsagePlans"
  } ]
}
```

For more information about migrating API stages associated with API keys, see Migrate to Default Usage Plans in the API Gateway Console.

Create a usage plan

The following procedure describes how to create a usage plan.

To create a usage plan with the REST API

1. Call `usageplan:create` to create a usage plan. In the payload, specify the name and description of the plan, associated API stages, rate limits, and quotas.

   Make note of the resultant usage plan identifier. You need it in the next step.

2. Do one of the following:

a. Call [usageplankey:create] to add an API key to the usage plan. Specify `keyId` and `keyType` in the payload.

   To add more API keys to the usage plan, repeat the previous call, one API key at a time.

b. Call [apikey:import] to add one or more API keys directly to the specified usage plan. The request payload should contain API key values, the associated usage plan identifier, the Boolean flags to indicate that the keys are enabled for the usage plan, and, possibly, the API key names and descriptions.

   The following example of the `apikey:import` request adds three API keys (as identified by `key`, `name`, and `description`) to one usage plan (as identified by `usageplanIds`):

```
POST /apikeys?mode=import&format=csv&failonwarnings=fase HTTP/1.1
Host: apigateway.us-east-1.amazonaws.com
Content-Type: text/csv
Authorization: ...

key,name, description, enabled, usageplanIds
abcdef1234ghijklmnop8901234567, importedKey_1, firstone,  tRuE, n371pt
abcdef1234ghijklmnop0123456789, importedKey_2, secondone, TRUE, n371pt
abcdef1234ghijklmnop9012345678, importedKey_3,          , true, n371pt
```

   As a result, three `UsagePlanKey` resources are created and added to the `UsagePlan`.

   You can also add API keys to more than one usage plan this way. To do this, change each `usageplanIds` column value to a comma-separated string that contains the selected usage plan identifiers, and is enclosed within a pair of quotes (`"n371pt,m282qs"` or `'n371pt,m282qs'`).

   Note

   An API key can be associated with more than one usage plan. A usage plan can be associated with more than one stage. However, a given API key can only be associated with one usage plan for each stage of your API.

Manage a usage plan by using the AWS CLI

The following code examples show how to add, remove, or modify the method-level throttling settings in a usage plan by calling the `update-usage-plan` command.

> **Note**
>
> Be sure to change `us-east-1` to the appropriate Region value for your API.

To add or replace a rate limit for throttling an individual resource and method:

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-
operations
                        op="replace",path="/apiStages/<apiId>:<stage>/
throttle/<resourcePath>/<httpMethod>/rateLimit",value="0.1"
```

To add or replace a burst limit for throttling an individual resource and method:

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId>
 --patch-operations op="replace",path="/apiStages/<apiId>:<stage>/
throttle/<resourcePath>/<httpMethod>/burstLimit",value="1"
```

To remove the method-level throttling settings for an individual resource and method:

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId>
 --patch-operations op="remove",path="/apiStages/<apiId>:<stage>/
throttle/<resourcePath>/<httpMethod>",value=""
```

To remove all method-level throttling settings for an API:

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-
operations op="remove",path="/apiStages/<apiId>:<stage>/throttle ",value=""
```

Here is an example using the Pet Store sample API:

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-
operations
```

```
                op="replace",path="/apiStages/<apiId>:<stage>/throttle",value='"{\"/
pets/GET\":{\"rateLimit\":1.0,\"burstLimit\":1},\"//GET\":{\"rateLimit\":1.0,
\"burstLimit\":1}}"'
```

Test usage plans

As an example, let's use the PetStore API, which was created in Tutorial: Create a REST API by importing an example. Assume that the API is configured to use an API key of `Hiorr45VR...c4GJc`. The following steps describe how to test a usage plan.

To test your usage plan

- Make a GET request on the Pets resource (`/pets`), with the `?type=...&page=...` query parameters, of the API (for example, `xbvxlpijch`) in a usage plan:

  ```
  GET /testStage/pets?type=dog&page=1 HTTP/1.1
  x-api-key: Hiorr45VR...c4GJc
  Content-Type: application/x-www-form-urlencoded
  Host: xbvxlpijch.execute-api.ap-southeast-1.amazonaws.com
  X-Amz-Date: 20160803T001845Z
  Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20160803/ap-southeast-1/
  execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date;x-api-key,
   Signature={sigv4_hash}
  ```

  > **Note**
  >
  > You must submit this request to the `execute-api` component of API Gateway and provide the required API key (for example, `Hiorr45VR...c4GJc`) in the required `x-api-key` header.

  The successful response returns a `200 OK` status code and a payload that contains the requested results from the backend. If you forget to set the `x-api-key` header or set it with an incorrect key, you get a `403 Forbidden` response. However, if you didn't configure the method to require an API key, you will likely get a `200 OK` response whether you set the `x-api-key` header correctly or not, and the throttle and quota limits of the usage plan are bypassed.

  Occasionally, when an internal error occurs where API Gateway is unable to enforce usage plan throttling limits or quotas for the request, API Gateway serves the request without applying the throttling limits or quotas as specified in the usage plan. But, it logs an error message of `Usage Plan check failed due to an internal error` in CloudWatch. You can ignore such occasional errors.

Create and configure API keys and usage plans with AWS CloudFormation

You can use AWS CloudFormation to require API keys on API methods and create a usage plan for an API. The example AWS CloudFormation template does the following:

- Creates an API Gateway API with GET and POST methods.
- Requires an API key for the GET and POST methods. This API receives keys from the X-API-KEY header of each incoming request.
- Creates an API key.
- Creates a usage plan to specify a monthly quota of 1,000 request each month, a throttling rate limit of 100 request each second, and a throttling burst limit of 200 request each second.
- Specifies a method-level throttling rate limit of 50 requests each second and a method-level throttling burst limit of 100 requests per second for the GET method.
- Associates the API stage and API key with the usage plan.

```
AWSTemplateFormatVersion: 2010-09-09
Parameters:
  StageName:
    Type: String
    Default: v1
    Description: Name of API stage.
  KeyName:
    Type: String
    Default: MyKeyName
    Description: Name of an API key
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: keys-api
      ApiKeySourceType: HEADER
  PetsResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'pets'
  PetsMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: GET
      ApiKeyRequired: true
      AuthorizationType: NONE
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  PetsMethodPost:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: POST
      ApiKeyRequired: true
      AuthorizationType: NONE
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - PetsMethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: !Sub '${StageName}'
  UsagePlan:
    Type: AWS::ApiGateway::UsagePlan
    DependsOn:
      - ApiDeployment
    Properties:
      Description: Example usage plan with a monthly quota of 1000 calls and method-
level throttling for /pets GET
      ApiStages:
        - ApiId: !Ref Api
          Stage: !Sub '${StageName}'
          Throttle:
            "/pets/GET":
              RateLimit: 50.0
              BurstLimit: 100
      Quota:
        Limit: 1000
        Period: MONTH
      Throttle:
        RateLimit: 100.0
        BurstLimit: 200
      UsagePlanName: "My Usage Plan"
  ApiKey:
    Type: AWS::ApiGateway::ApiKey
    Properties:
      Description: API Key
      Name: !Sub '${KeyName}'
      Enabled: True
  UsagePlanKey:
    Type: AWS::ApiGateway::UsagePlanKey
    Properties:
      KeyId: !Ref ApiKey
      KeyType: API_KEY
      UsagePlanId: !Ref UsagePlan
Outputs:
  ApiRootUrl:
    Description: Root Url of the API
    Value: !Sub 'https://${Api}.execute-api.${AWS::Region}.amazonaws.com/${StageName}'
```

API Gateway API key file format

API Gateway can import API keys from external files of a comma-separated value (CSV) format, and then associate the imported keys with one or more usage plans. The imported file must contain the `Name` and `Key` columns. The column header names aren't case sensitive, and columns can be in any order, as shown in the following example:

```
Key,name
apikey1234abcdefghij0123456789,MyFirstApiKey
```

A `Key` value must be between 20 and 128 characters. A `Name` value cannot exceed 1024 characters.

An API key file can also have the `Description`, `Enabled`, or `UsagePlanIds` column, as shown in the following example:

```
Name,key,description,Enabled,usageplanIds
MyFirstApiKey,apikey1234abcdefghij0123456789,An imported key,TRUE,c7y23b
```

When a key is associated with more than one usage plan, the `UsagePlanIds` value is a comma-separated string of the usage plan IDs, enclosed with a pair of double or single quotes, as shown in the following example:

```
Enabled,Name,key,UsageplanIds
true,MyFirstApiKey,apikey1234abcdefghij0123456789,"c7y23b,glvrsr"
```

Unrecognized columns are permitted, but are ignored. The default value is an empty string or a `true` Boolean value.

The same API key can be imported multiple times, with the most recent version overwriting the previous one. Two API keys are identical if they have the same `key` value.

> **Note**
>
> For best practices to consider, see the section called “Best practices for API keys and usage plans”.

Documenting REST APIs

To help customers understand and use your API, you should document the API. To help you document your API, API Gateway lets you add and update the help content for individual API entities as an integral part of your API development process. API Gateway stores the source content and enables you to archive different versions of the documentation. You can associate a documentation version with an API stage, export a stage-specific documentation snapshot to an external OpenAPI file, and distribute the file as a publication of the documentation.

To document your API, you can call the API Gateway REST API, use one of the AWS SDKs or AWS CLIs for API Gateway, or use the API Gateway console. In addition, you can import or export the documentation parts that are defined in an external OpenAPI file.

To share API documentation with developers, you can use a developer portal. For an example, see Integrating ReadMe with API Gateway to Keep Your Developer Hub Up to Date on the AWS Partner Network (APN) blog.

Topics

Representation of API documentation in API Gateway

API Gateway API documentation consists of individual documentation parts associated with specific API entities that include API, resource, method, request, response, message parameters (i.e., path, query, header), as well as authorizers and models.

In API Gateway, a documentation part is represented by a DocumentationPart resource. The API documentation as a whole is represented by the DocumentationParts collection.

Documenting an API involves creating `DocumentationPart` instances, adding them to the `DocumentationParts` collection, and maintaining versions of the documentation parts as your API evolves.

Topics

Documentation parts

A DocumentationPart resource is a JSON object that stores the documentation content applicable to an individual API entity. Its `properties` field contains the documentation content as a map of key-value pairs. Its `location` property identifies the associated API entity.

The shape of a content map is determined by you, the API developer. The value of a key-value pair can be a string, number, boolean, object, or array. The shape of the `location` object depends on the targeted entity type.

The DocumentationPart resource supports content inheritance: the documentation content of an API entity is applicable to children of that API entity. For more information about the definition of child entities and content inheritance, see Inherit Content from an API Entity of More General Specification.

Location of a documentation part

The location property of a DocumentationPart instance identifies an API entity to which the associated content applies. The API entity can be an API Gateway REST API resource, such as RestApi, Resource, Method, MethodResponse, Authorizer, or Model. The entity can also be a message parameter, such as a URL path parameter, a query string parameter, a request or response header parameter, a request or response body, or response status code.

To specify an API entity, set the type attribute of the location object to be one of API, AUTHORIZER, MODEL, RESOURCE, METHOD, PATH_PARAMETER, QUERY_PARAMETER, REQUEST_HEADER, REQUEST_BODY, RESPONSE, RESPONSE_HEADER, or RESPONSE_BODY.

Depending on the type of an API entity, you might specify other location attributes, including method, name, path, and statusCode. Not all of these attributes are valid for a given API entity. For example, type, path, name, and statusCode are valid attributes of the RESPONSE entity; only type and path are valid location attributes of the RESOURCE entity. It is an error to include an invalid field in the location of a DocumentationPart for a given API entity.

Not all valid location fields are required. For example, type is both the valid and required location field of all API entities. However, method, path, and statusCode are valid but not required attributes for the RESPONSE entity. When not explicitly specified, a valid location field assumes its default value. The default path value is /, i.e., the root resource of an API. The default value of method, or statusCode is *, meaning any method, or status code values, respectively.

Content of a documentation part

The properties value is encoded as a JSON string. The properties value contains any information you choose to meet your documentation requirements. For example, the following is a valid content map:

```
{
  "info": {
    "description": "My first API with Amazon API Gateway."
  },
  "x-custom-info" : "My custom info, recognized by OpenAPI.",
  "my-info" : "My custom info not recognized by OpenAPI."
}
```

Although API Gateway accepts any valid JSON string as the content map, the content attributes are treated as two categories: those that can be recognized by OpenAPI and those that cannot. In the preceding example, `info`, `description`, and `x-custom-info` are recognized by OpenAPI as a standard OpenAPI object, property, or extension. In contrast, `my-info` is not compliant with the OpenAPI specification. API Gateway propagates OpenAPI-compliant content attributes into the API entity definitions from the associated `DocumentationPart` instances. API Gateway does not propagate the non-compliant content attributes into the API entity definitions.

As another example, here is `DocumentationPart` targeted for a `Resource` entity:

```
{
    "location" : {
        "type" : "RESOURCE",
        "path": "/pets"
    },
    "properties" : {
        "summary" : "The /pets resource represents a collection of pets in PetStore.",
        "description": "... a child resource under the root...",
    }
}
```

Here, both `type` and `path` are valid fields to identify the target of the `RESOURCE` type. For the root resource (/), you can omit the `path` field.

```
{
    "location" : {
        "type" : "RESOURCE"
    },
    "properties" : {
        "description" : "The root resource with the default path specification."
    }
}
```

This is the same as the following `DocumentationPart` instance:

```
{
    "location" : {
        "type" : "RESOURCE",
        "path": "/"
    },
    "properties" : {
        "description" : "The root resource with an explicit path specification"
    }
}
```

Inherit content from an API entity of more general specifications

The default value of an optional `location` field provides a patterned description of an API entity. Using the default value in the `location` object, you can add a general description in the `properties` map to a `DocumentationPart` instance with this type of `location` pattern. API Gateway extracts the applicable OpenAPI documentation attributes from the `DocumentationPart` of the generic API entity and injects them into a specific API entity with the `location` fields matching the general `location` pattern, or matching the exact value, unless the specific entity already has a `DocumentationPart` instance associated with it. This behavior is also known as content inheritance from an API entity of more general specifications.

Content inheritance does not apply to certain API entity types. See the table below for details.

When an API entity matches more than one `DocumentationPart`'s location pattern, the entity will inherit the documentation part with the location fields of the highest precedence and specificities. The order of precedence is `path` > `statusCode`. For matching with the `path` field, API Gateway chooses the entity with the most specific path value. The following table shows this with a few examples.

Case	**path**	**statusCode**	**name**	**Remarks**
1	`/pets`	`*`	`id`	Documentation associated with this location pattern will be

Case	path	statusCode	name	Remar
				inherited by entities matching the location pattern.

Case	path	statusCode	name	Remar
2	/pets	200	id	Documentation associated with this location pattern will be inherited by entities matching the location pattern when both Case 1 and Case 2 are matched, because Case 2 is more specific than

Case	path	statusCo e	name	Remar
				Case 1.

Case	path	statusCode	name	Remarks
3	/pets/petId	*	id	Documentation associated with this location pattern will be inherited by entities matching the location pattern when Cases 1, 2, and 3 are matched, because Case 3 has a higher precedence than

Case	path	statusCode	name	Remar
				Case 2 and is more specific than Case 1.

Here is another example to contrast a more generic `DocumentationPart` instance to a more specific one. The following general error message of "`Invalid request error`" is injected into the OpenAPI definitions of the `400` error responses, unless overridden.

```
{
    "location" : {
        "type" : "RESPONSE",
        "statusCode": "400"
    },
    "properties" : {
        "description" : "Invalid request error."
    }"
}
```

With the following overwrite, the `400` responses to any methods on the `/pets` resource has a description of "`Invalid petId specified`" instead.

```
{
    "location" : {
        "type" : "RESPONSE",
        "path": "/pets",
        "statusCode": "400"
    },
    "properties" : "{
        "description" : "Invalid petId specified."
    }"
}
```

Valid location fields of `DocumentationPart`

The following table shows the valid and required fields as well as applicable default values of a DocumentationPart resource that is associated with a given type of API entities.

API entity	Valid location fields	Required location fields	Default field values	Inheritable content

API	`{`
	`    "location": {`
	`        "type": "API"`
	`    },`
	`    ...`
	`}`
		`type`	N/A	No

Resource	`{`
	`    "location": {`
	`        "type": "RESOURCE`
	`",`
	`        "path":`
	` "resource_path "`
	`    },`
	`    ...`
	`}`
		`type`	The default value of `path` is `/`.	No

Method	`{`
	`    "location": {`
	`        "type":`
	` "METHOD",`
	`        "path":`
	` "resource_path ",`
	`        "method":`
	` "http_verb "`
	`    },`
	`    ...`
	`}`
		`type`	The default values of `path` and `method` are `/` and `*`, respectively.	Yes, matching `path` by prefix and matching `method` of any values.

API entity	Valid location fields	Required location fields	Default field values	Inheritable content

Query parameter	`{`
	`    "location": {`
	`        "type": "QUERY_PARAMETER",`
	`        "path": "resource_path ",`
	`        "method": "HTTP_verb ",`
	`        "name": "query_parameter_name "`
	`    },`
	`    ...`
	`}`
		`type`	The default values of `path` and `method` are / and `*`, respectively.	Yes, matching `path` by prefix and matching `method` by exact values.

Request body	`{`
	`    "location": {`
	`        "type": "REQUEST_BODY",`
	`        "path": "resource_path ",`
	`        "method": "http_verb "`
	`    },`
	`    ...`
	`}`
		`type`	The default values of `path`, and `method` are /and `*`, respectively.	Yes, matching `path` by prefix, and matching `method` by exact values.

API entity	Valid location fields	Required location fields	Default field values	Inheritable content
Request header parameter	`{ "location": { "type": "REQUEST_HEADER", "path": "resource_path ", "method": "HTTP_verb ", "name": "header_name " }, ... }`	`type`, `name`	The default values of `path` and `method` are `/` and `*`, respectively.	Yes, matching `path` by prefix and matching `method` by exact values.
Request path parameter	`{ "location": { "type": "PATH_PARAMETER", "path": "resource/{path_parameter_name }", "method": "HTTP_verb ", "name": "path_parameter_name " }, ... }`	`type`, `name`	The default values of `path` and `method` are `/` and `*`, respectively.	Yes, matching `path` by prefix and matching `method` by exact values.

API entity	Valid location fields	Required location fields	Default field values	Inheritable content

Response	`{`
	`    "location": {`
	`        "type": "RESPONSE",`
	`        "path": "resource_path ",`
	`        "method": "http_verb ",`
	`        "statusCode": "status_code "`
	`    },`
	`    ...`
	`}`
		`type`	The default values of `path`, `method`, and `statusCode` are /, * and *, respectively.	Yes, matching `path` by prefix and matching `method` and `statusCode` by exact values.

Response header	`{`
	`    "location": {`
	`        "type": "RESPONSE_HEADER",`
	`        "path": "resource_path ",`
	`        "method": "http_verb ",`
	`        "statusCode": "status_code ",`
	`        "name": "header_name "`
	`    },`
	`    ...`
	`}`
		`type`, `name`	The default values of `path`, `method` and `statusCode` are /, * and *, respectively.	Yes, matching `path` by prefix and matching `method`, and `statusCode` by exact values.

<table>
<tr><th>API entity</th><th>Valid location fields</th><th>Required location fields</th><th>Default field values</th><th>Inheritable content</th></tr>
<tr><td>Response body</td><td><pre>{
    "location": {
        "type": "RESPONSE_BODY",
        "path": "<i>resource_path</i> ",
        "method": "<i>http_verb</i> ",
        "statusCode": "<i>status_code</i> "
    },
    ...
}</pre></td><td><code>type</code></td><td>The default values of <code>path</code>, <code>method</code> and <code>statusCode</code> are /, * and *, respectively.</td><td>Yes, matching <code>path</code> by prefix and matching <code>method</code>, and <code>statusCode</code> by exact values.</td></tr>
<tr><td>Authorizer</td><td><pre>{
    "location": {
        "type": "AUTHORIZER",
        "name": "<i>authorizer_name</i> "
    },
    ...
}</pre></td><td><code>type</code></td><td>N/A</td><td>No</td></tr>
<tr><td>Model</td><td><pre>{
    "location": {
        "type": "MODEL",
        "name": "<i>model_name</i> "
    },
    ...
}</pre></td><td><code>type</code></td><td>N/A</td><td>No</td></tr>
</table>

Documentation versions

A documentation version is a snapshot of the DocumentationParts collection of an API and is tagged with a version identifier. Publishing the documentation of an API involves creating a documentation version, associating it with an API stage, and exporting that stage-specific version of the API documentation to an external OpenAPI file. In API Gateway, a documentation snapshot is represented as a DocumentationVersion resource.

As you update an API, you create new versions of the API. In API Gateway, you maintain all the documentation versions using the DocumentationVersions collection.

Document an API using the API Gateway console

In this section, we describe how to create and maintain documentation parts of an API using the API Gateway console.

A prerequisite for creating and editing the documentation of an API is that you must have already created the API. In this section, we use the PetStore API as an example. To create an API using the API Gateway console, follow the instructions in Tutorial: Create a REST API by importing an example.

Topics

Document the API entity

To add a new documentation part for the `API` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **API**.

   If a documentation part was not created for the `API`, you get the documentation part's `properties` map editor. Enter the following `properties` map in the text editor.

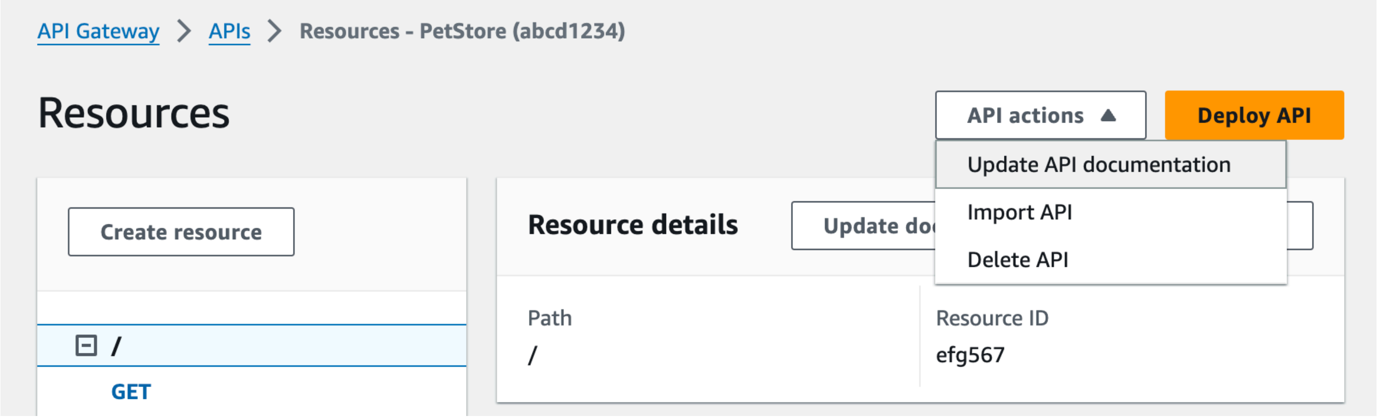
   ```
   {
     "info": {
       "description": "Your first API Gateway API.",
       "contact": {
           "name": "John Doe",
           "email": "john.doe@api.com"
       }
     }
   }
   ```

   > **Note**
   >
   > You do not need to encode the `properties` map into a JSON string. The API Gateway console stringifies the JSON object for you.

3. Choose **Create documentation part**.

To add a new documentation part for the `API` entity in the **Resources** pane, do the following:

1. In the main navigation pane, choose **Resources**.
2. Choose the **API actions** menu, and then choose **Update API documentation**.



To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. Select the name of your API, and then on the API card, choose **Edit**.

Document a RESOURCE entity

To add a new documentation part for a RESOURCE entity, do the following:

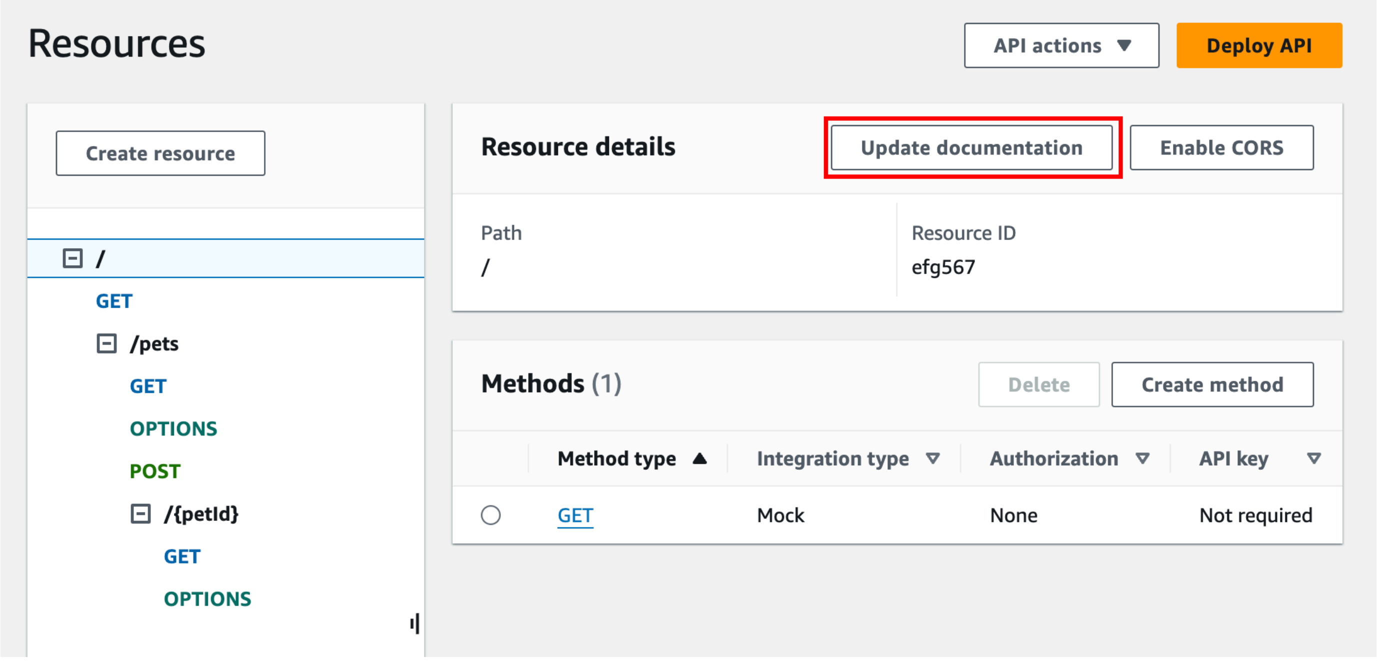
1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Resource**.
3. For **Path**, enter a path.
4. Enter a description in the text editor, for example:

```
{
    "description": "The PetStore's root resource."
}
```

5. Choose **Create documentation part**. You can create documentation for an unlisted resource.
6. If required, repeat these steps to add or edit another documentation part.

To add a new documentation part for a RESOURCE entity in the **Resources** pane, do the following:

1. In the main navigation pane, choose **Resources**.
2. Choose the resource, and then choose **Update documentation**.



To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. Select the resource containing your documentation part, and then choose **Edit**.

Document a `METHOD` entity

To add a new documentation part for a `METHOD` entity, do the following:

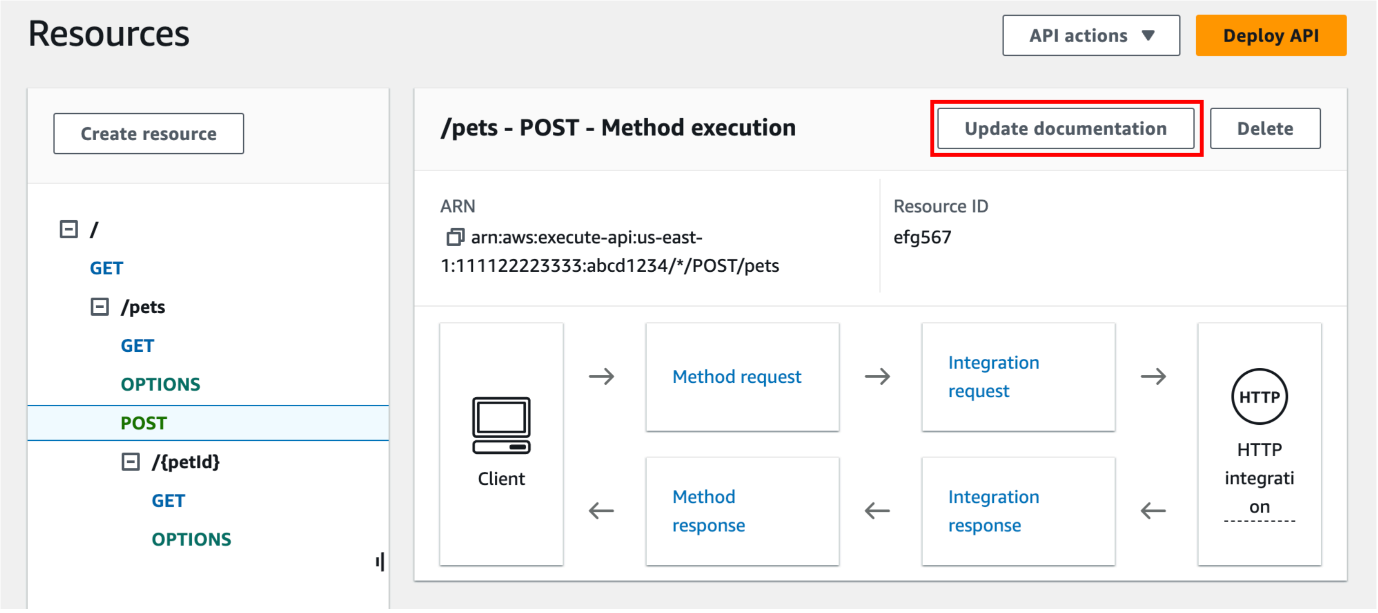
1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Method**.
3. For **Path**, enter a path.
4. For **Method**, select an HTTP verb.
5. Enter a description in the text editor, for example:

```
{
  "tags" : [ "pets" ],
  "summary" : "List all pets"
}
```

6. Choose **Create documentation part**. You can create documentation for an unlisted method.
7. If required, repeat these steps to add or edit another documentation part.

To add a new documentation part for a `METHOD` entity in the **Resources** pane, do the following:

1. In the main navigation pane, choose **Resources**.
2. Choose the method, and then choose **Update documentation**.



ARN
arn:aws:execute-api:us-east-1:111122223333:abcd1234/*/POST/pets

Resource ID
efg567

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the method or select the resource containing the method, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `QUERY_PARAMETER` entity

To add a new documentation part for a `QUERY_PARAMETER` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Query parameter**.

3. For **Path**, enter a path.
4. For **Method**, select an HTTP verb.
5. For **Name**, enter a name.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted query parameter.
8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the query parameter or select the resource containing the query parameter, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `PATH_PARAMETER` entity

To add a new documentation part for a `PATH_PARAMETER` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Path parameter**.
3. For **Path**, enter a path.
4. For **Method**, select an HTTP verb.
5. For **Name**, enter a name.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted path parameter.
8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.

2. You can select the path parameter or select the resource containing the path parameter, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `REQUEST_HEADER` entity

To add a new documentation part for a `REQUEST_HEADER` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Request header**.
3. For **Path**, enter a path for the request header.
4. For **Method**, select an HTTP verb.
5. For **Name**, enter a name.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted request header.
8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the request header or select the resource containing the request header, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `REQUEST_BODY` entity

To add a new documentation part for a `REQUEST_BODY` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Request body**.
3. For **Path**, enter a path for the request body.

4. For **Method**, select an HTTP verb.
5. Enter a description in the text editor.
6. Choose **Create documentation part**. You can create documentation for an unlisted request body.
7. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the request body or select the resource containing the request body, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `RESPONSE` entity

To add a new documentation part for a `RESPONSE` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Response (status code)**.
3. For **Path**, enter a path for the response.
4. For **Method**, select an HTTP verb.
5. For **Status code**, enter an HTTP status code.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted response status code.
8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the response status code or select the resource containing the response status code, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `RESPONSE_HEADER` entity

To add a new documentation part for a `RESPONSE_HEADER` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Response header**.
3. For **Path**, enter a path for the response header.
4. For **Method**, select an HTTP verb.
5. For **Status code**, enter an HTTP status code.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted response header.
8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the response header or select the resource containing the response header, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `RESPONSE_BODY` entity

To add a new documentation part for a `RESPONSE_BODY` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Response body**.
3. For **Path**, enter a path for the response body.
4. For **Method**, select an HTTP verb.
5. For **Status code**, enter an HTTP status code.
6. Enter a description in the text editor.
7. Choose **Create documentation part**. You can create documentation for an unlisted response body.

8. If required, repeat these steps to add or edit another documentation part.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Resources and methods** tab.
2. You can select the response body or select the resource containing the response body, and then use the search bar to find and select your documentation part.
3. Choose **Edit**.

Document a `MODEL` entity

Documenting a `MODEL` entity involves creating and managing `DocumentPart` instances for the model and each of the model's `properties`'. For example, for the `Error` model that comes with every API by default has the following schema definition,

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "Error Schema",
  "type" : "object",
  "properties" : {
    "message" : { "type" : "string" }
  }
}
```

and requires two `DocumentationPart` instances, one for the `Model` and the other for its `message` property:

```
{
  "location": {
    "type": "MODEL",
    "name": "Error"
  },
  "properties": {
    "title": "Error Schema",
    "description": "A description of the Error model"
  }
}
```

and

```
{
  "location": {
    "type": "MODEL",
    "name": "Error.message"
  },
  "properties": {
    "description": "An error message."
  }
}
```

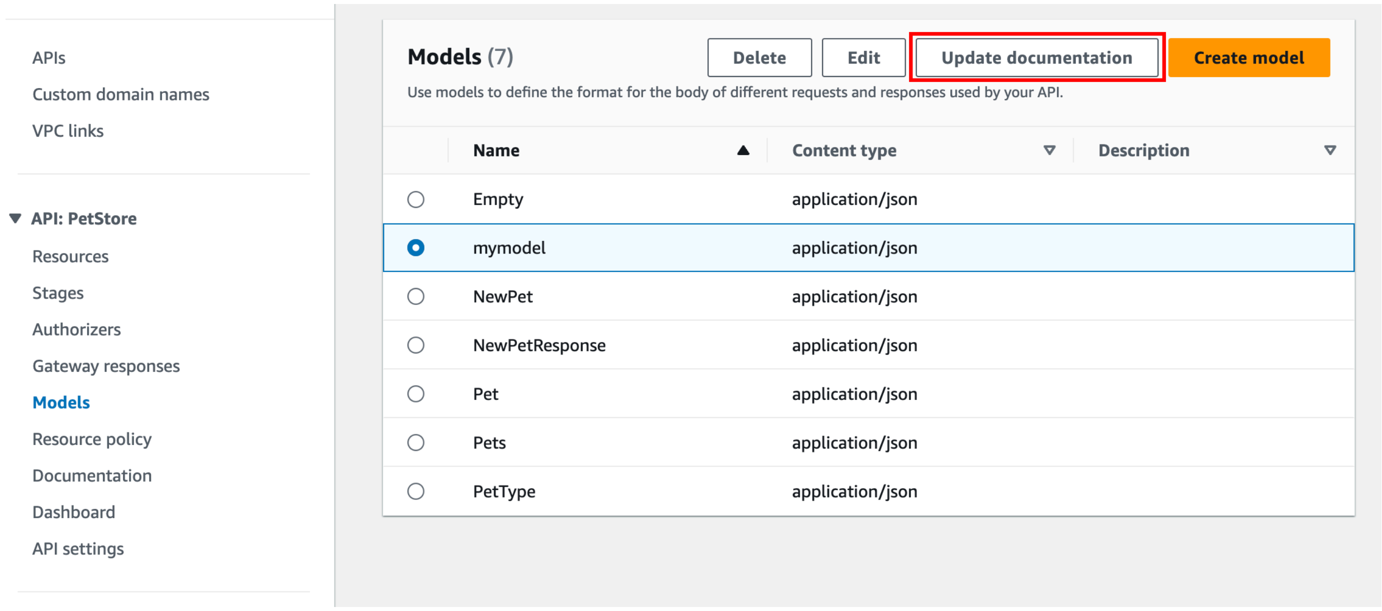
When the API is exported, the `DocumentationPart`'s properties will override the values in the original schema.

To add a new documentation part for a `MODEL` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Model**.
3. For **Name**, enter a name for the model.
4. Enter a description in the text editor.
5. Choose **Create documentation part**. You can create documentation for unlisted models.
6. If required, repeat these steps to add or edit a documentation part to other models.

To add a new documentation part for a `MODEL` entity in the **Models** pane, do the following:

1. In the main navigation pane, choose **Models**.
2. Choose the model, and then choose **Update documentation**.



APIs
Custom domain names
VPC links

▼ API: PetStore
Resources
Stages
Authorizers
Gateway responses
Models
Resource policy
Documentation
Dashboard
API settings

Models (7)

Use models to define the format for the body of different requests and responses used by your API.

	Name	Content type	Description
○	Empty	application/json	
●	mymodel	application/json	
○	NewPet	application/json	
○	NewPetResponse	application/json	
○	Pet	application/json	
○	Pets	application/json	
○	PetType	application/json	

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Models** tab.
2. Use the search bar or select the model, and then choose **Edit**.

Document an `AUTHORIZER` entity

To add a new documentation part for an `AUTHORIZER` entity, do the following:

1. In the main navigation pane, choose **Documentation**, and then choose **Create documentation part**.
2. For **Documentation type**, select **Authorizer**.
3. For **Name**, enter the name of your authorizer.
4. Enter a description in the text editor. Specify a value for the valid `location` field for the authorizer.
5. Choose **Create documentation part**. You can create documentation for unlisted authorizers.
6. If required, repeat these steps to add or edit a documentation part to other authorizers.

To edit an existing documentation part, do the following:

1. In the **Documentation** pane, choose the **Authorizers** tab.

2. Use the search bar or select the authorizer, and then choose **Edit**.

Publish API documentation using the API Gateway console

The following procedure describes how to publish a documentation version.

To publish a documentation version using the API Gateway console

1. In the main navigation pane, choose **Documentation**.
2. Choose **Publish documentation**.
3. Set up the publication:

   a. For **Stage**, select a stage.

   b. For **Version**, enter a version identifier, e.g., `1.0.0`.

   c. (Optional) For **Description**, enter a description.
4. Choose **Publish**.

You can now proceed to download the published documentation by exporting the documentation to an external OpenAPI file. To learn more, see the section called "Export a REST API".

Document an API using the API Gateway REST API

In this section, we describe how to create and maintain documentation parts of an API using the API Gateway REST API.

Before creating and editing the documentation of an API, first create the API. In this section, we use the PetStore API as an example. To create an API using the API Gateway console, follow the instructions in Tutorial: Create a REST API by importing an example.

Topics

- Document a REQUEST_HEADER entity
- Document a RESPONSE entity
- Document a RESPONSE_HEADER entity
- Document an AUTHORIZER entity
- Document a MODEL entity
- Update documentation parts
- List documentation parts

Document the API entity

To add documentation for an API, add a DocumentationPart resource for the API entity:

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "API"
    },
    "properties": "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon
 API Gateway.\"\n\t}\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  ...
  "id": "s2e5xf",
  "location": {
    "path": null,
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "API"
  },
  "properties": "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon
 API Gateway.\"\n\t}\n}"
}
```

If the documentation part has already been added, a `409 Conflict` response returns, containing the error message of `Documentation part already exists for the specified location: type 'API'."` In this case, you must call the documentationpart:update operation.

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/properties",
    "value" : "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon API
 Gateway.\"\n\t}\n}"
  } ]
}
```

The successful response returns a `200 OK` status code with the payload containing the updated `DocumentationPart` instance in the payload.

Document a RESOURCE entity

To add documentation for the root resource of an API, add a DocumentationPart resource targeted for the corresponding Resource resource:

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

```
{
    "location" : {
       "type" : "RESOURCE",
    },
    "properties" : "{\n\t\"description\" : \"The PetStore root resource.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    }
  },
  "id": "p76vqo",
  "location": {
    "path": "/",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
  "properties": "{\n\t\"description\" : \"The PetStore root resource.\"\n}"
}
```

When the resource path is not specified, the resource is assumed to be the root resource. You can add `"path": "/"` to `properties` to make the specification explicit.

To create documentation for a child resource of an API, add a DocumentationPart resource targeted for the corresponding Resource resource:

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "RESOURCE",
         "path" : "/pets"
    },
    "properties": "{\n\t\"description\" : \"A child resource under the root of
 PetStore.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    }
  },
  "id": "qcht86",
  "location": {
    "path": "/pets",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
  "properties": "{\n\t\"description\" : \"A child resource under the root of PetStore.
\"\n}"
}
```

To add documentation for a child resource specified by a path parameter, add a DocumentationPart resource targeted for the Resource resource:

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "RESOURCE",
         "path" : "/pets/{petId}"
    },
    "properties": "{\n\t\"description\" : \"A child resource specified by the petId
 path parameter.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    }
  },
  "id": "k6fpwb",
  "location": {
    "path": "/pets/{petId}",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
  "properties": "{\n\t\"description\" : \"A child resource specified by the petId path
 parameter.\"\n}"
}
```

Note

The DocumentationPart instance of a RESOURCE entity cannot be inherited by any of its child resources.

Document a METHOD entity

To add documentation for a method of an API, add a DocumentationPart resource targeted for the corresponding Method resource:

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "METHOD",
         "path" : "/pets",
         "method" : "GET"
    },
    "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

If the `location.method` field is not specified in the preceding request, it is assumed to be ANY method that is represented by a wild card `*` character.

To update the documentation content of a `METHOD` entity, call the [documentationpart:update](https://docs.aws.amazon.com/apigateway/latest/api/API_UpdateDocumentationPart.html) operation, supplying a new `properties` map:

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

```
{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/properties",
    "value" : "{\n\t\"tags\" : [ \"pets\" ], \n\t\"summary\" : \"List all pets.\"\n}"
  } ]
}
```

The successful response returns a `200 OK` status code with the payload containing the updated `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"tags\" : [ \"pets\" ], \n\t\"summary\" : \"List all pets.\"\n}"
}
```

Document a QUERY_PARAMETER entity

To add documentation for a request query parameter, add a DocumentationPart resource targeted for the QUERY_PARAMETER type, with the valid fields of path and name.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "QUERY_PARAMETER",
         "path" : "/pets",
         "method" : "GET",
         "name" : "page"
    },
    "properties": "{\n\t\"description\" : \"Page number of results to return.\"\n}"
}
```

If successful, the operation returns a 201 Created response containing the newly created DocumentationPart instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    }
  },
  "id": "h9ht5w",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": "page",
    "statusCode": null,
    "type": "QUERY_PARAMETER"
  },
  "properties": "{\n\t\"description\" : \"Page number of results to return.\"\n}"
}
```

The documentation part's `properties` map of a `QUERY_PARAMETER` entity can be inherited by one of its child `QUERY_PARAMETER` entities. For example, if you add a `treats` resource after `/pets/{petId}`, enable the `GET` method on `/pets/{petId}/treats`, and expose the `page` query parameter, this child query parameter inherits the `DocumentationPart's` `properties` map from the like-named query parameter of the `GET /pets` method, unless you explicitly add a `DocumentationPart` resource to the `page` query parameter of the `GET /pets/{petId}/treats` method.

Document a PATH_PARAMETER entity

To add documentation for a path parameter, add a [DocumentationPart](#) resource for the `PATH_PARAMETER` entity.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "PATH_PARAMETER",
         "path" : "/pets/{petId}",
         "method" : "*",
         "name" : "petId"
    },
    "properties": "{\n\t\"description\" : \"The id of the pet to retrieve.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    }
  },
  "id": "ckpgog",
  "location": {
    "path": "/pets/{petId}",
    "method": "*",
    "name": "petId",
    "statusCode": null,
    "type": "PATH_PARAMETER"
  },
  "properties": "{\n  \"description\" : \"The id of the pet to retrieve\"\n}"
}
```

Document a REQUEST_BODY entity

To add documentation for a request body, add a DocumentationPart resource for the request body.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date, Signature=sigv4_secret
```

```
{
    "location" : {
         "type" : "REQUEST_BODY",
         "path" : "/pets",
         "method" : "POST"
    },
    "properties": "{\n\t\"description\" : \"A Pet object to be added to PetStore.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    }
  },
  "id": "kgmfr1",
  "location": {
    "path": "/pets",
    "method": "POST",
    "name": null,
    "statusCode": null,
    "type": "REQUEST_BODY"
  },
  "properties": "{\n\t\"description\" : \"A Pet object to be added to PetStore.\"\n}"
}
```

Document a REQUEST_HEADER entity

To add documentation for a request header, add a DocumentationPart resource for the request header.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "REQUEST_HEADER",
         "path" : "/pets",
         "method" : "GET",
         "name" : "x-my-token"
    },
    "properties": "{\n\t\"description\" : \"A custom token used to authorization the
 method invocation.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    }
  },
  "id": "h0m3uf",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": "x-my-token",
    "statusCode": null,
    "type": "REQUEST_HEADER"
  },
  "properties": "{\n\t\"description\" : \"A custom token used to authorization the
 method invocation.\"\n}"
}
```

Document a RESPONSE entity

To add documentation for a response of a status code, add a DocumentationPart resource targeted for the corresponding MethodResponse resource.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location": {
      "path": "/",
      "method": "*",
      "name": null,
      "statusCode": "200",
      "type": "RESPONSE"
    },
    "properties": "{\n  \"description\" : \"Successful operation.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
    "_links": {
        "self": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        }
    },
    "id": "lattew",
    "location": {
        "path": "/",
        "method": "*",
        "name": null,
        "statusCode": "200",
        "type": "RESPONSE"
    },
    "properties": "{\n  \"description\" : \"Successful operation.\"\n}"
}
```

Document a RESPONSE_HEADER entity

To add documentation for a response header, add a DocumentationPart resource for the response header.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

  "location": {
    "path": "/",
    "method": "GET",
    "name": "Content-Type",
    "statusCode": "200",
    "type": "RESPONSE_HEADER"
  },
  "properties": "{\n  \"description\" : \"Media type of request\"\n}"
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    }
  },
  "id": "fev7j7",
  "location": {
    "path": "/",
    "method": "GET",
    "name": "Content-Type",
    "statusCode": "200",
    "type": "RESPONSE_HEADER"
  },
  "properties": "{\n  \"description\" : \"Media type of request\"\n}"
}
```

The documentation of this `Content-Type` response header is the default documentation for the `Content-Type` headers of any responses of the API.

Document an `AUTHORIZER` entity

To add documentation for an API authorizer, add a DocumentationPart resource targeted for the specified authorizer.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
```

```
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "AUTHORIZER",
         "name" : "myAuthorizer"
    },
    "properties": "{\n\t\"description\" : \"Authorizes invocations of configured
 methods.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    }
  },
  "id": "pw3qw3",
  "location": {
    "path": null,
    "method": null,
    "name": "myAuthorizer",
    "statusCode": null,
    "type": "AUTHORIZER"
  },
  "properties": "{\n\t\"description\" : \"Authorizes invocations of configured methods.
\"\n}"
}
```

Note

The DocumentationPart instance of an `AUTHORIZER` entity cannot be inherited by any of its child resources.

Document a `MODEL` entity

Documenting a `MODEL` entity involves creating and managing `DocumentPart` instances for the model and each of the model's `properties`'. For example, for the `Error` model that comes with every API by default has the following schema definition,

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "Error Schema",
  "type" : "object",
  "properties" : {
    "message" : { "type" : "string" }
  }
}
```

and requires two `DocumentationPart` instances, one for the `Model` and the other for its `message` property:

```
{
  "location": {
    "type": "MODEL",
    "name": "Error"
  },
  "properties": {
    "title": "Error Schema",
    "description": "A description of the Error model"
  }
}
```

and

```
{
  "location": {
    "type": "MODEL",
    "name": "Error.message"
  },
  "properties": {
    "description": "An error message."
  }
}
```

When the API is exported, the `DocumentationPart`'s properties will override the values in the original schema.

To add documentation for an API model, add a DocumentationPart resource targeted for the specified model.

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "MODEL",
         "name" : "Pet"
    },
    "properties": "{\n\t\"description\" : \"Data structure of a Pet object.\"\n}"
}
```

If successful, the operation returns a `201 Created` response containing the newly created `DocumentationPart` instance in the payload. For example:

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    }
  },
  "id": "lkn4uq",
  "location": {
    "path": null,
    "method": null,
    "name": "Pet",
    "statusCode": null,
    "type": "MODEL"
  },
  "properties": "{\n\t\"description\" : \"Data structure of a Pet object.\"\n}"
}
```

Repeat the same step to create a DocumentationPart instance for any of the model's properties.

Note

The DocumentationPart instance of a `MODEL` entity cannot be inherited by any of its child resources.

Update documentation parts

To update the documentation parts of any type of API entities, submit a PATCH request on a DocumentationPart instance of a specified part identifier to replace the existing `properties` map with a new one.

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
```

```
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "RESOURCE_PATH",
    "value" : "NEW_properties_VALUE_AS_JSON_STRING"
  } ]
}
```

The successful response returns a `200 OK` status code with the payload containing the updated `DocumentationPart` instance in the payload.

You can update multiple documentation parts in a single PATCH request.

List documentation parts

To list the documentation parts of any type of API entities, submit a GET request on a DocumentationParts collection.

```
GET /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

The successful response returns a `200 OK` status code with the payload containing the available `DocumentationPart` instances in the payload.

Publish API documentation using the API Gateway REST API

To publish the documentation for an API, create, update, or get a documentation snapshot, and then associate the documentation snapshot with an API stage. When creating a documentation snapshot, you can also associate it with an API stage at the same time.

Topics

- Create a documentation snapshot and associate it with an API stage

- Create a documentation snapshot
- Update a documentation snapshot
- Get a documentation snapshot
- Associate a documentation snapshot with an API stage
- Download a documentation snapshot associated with a stage

Create a documentation snapshot and associate it with an API stage

To create a snapshot of an API's documentation parts and associate it with an API stage at the same time, submit the following POST request:

```
POST /restapis/restapi_id/documentation/versions HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "documentationVersion" : "1.0.0",
    "stageName": "prod",
    "description" : "My API Documentation v1.0.0"
}
```

If successful, the operation returns a 200 OK response, containing the newly created DocumentationVersion instance as the payload.

Alternatively, you can create a documentation snapshot without associating it with an API stage first and then call restapi:update to associate the snapshot with a specified API stage. You can also update or query an existing documentation snapshot and then update its stage association. We show the steps in the next four sections.

Create a documentation snapshot

To create a snapshot of an API's documentation parts, create a new DocumentationVersion resource and add it to the DocumentationVersions collection of the API:

```
POST /restapis/restapi_id/documentation/versions HTTP/1.1
```

```
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "documentationVersion" : "1.0.0",
    "description" : "My API Documentation v1.0.0"
}
```

If successful, the operation returns a `200 OK` response, containing the newly created `DocumentationVersion` instance as the payload.

Update a documentation snapshot

You can only update a documentation snapshot by modifying the `description` property of the corresponding DocumentationVersion resource. The following example shows how to update the description of the documentation snapshot as identified by its version identifier, *`version`*, e.g., `1.0.0`.

```
PATCH /restapis/restapi_id/documentation/versions/version HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "patchOperations": [{
        "op": "replace",
        "path": "/description",
        "value": "My API for testing purposes."
    }]
}
```

If successful, the operation returns a `200 OK` response, containing the updated `DocumentationVersion` instance as the payload.

Get a documentation snapshot

To get a documentation snapshot, submit a GET request against the specified DocumentationVersion resource. The following example shows how to get a documentation snapshot of a given version identifier, 1.0.0.

```
GET /restapis/<restapi_id>/documentation/versions/1.0.0 HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

Associate a documentation snapshot with an API stage

To publish the API documentation, associate a documentation snapshot with an API stage. You must have already created an API stage before associating the documentation version with the stage.

To associate a documentation snapshot with an API stage using the API Gateway REST API, call the stage:update operation to set the desired documentation version on the `stage.documentationVersion` property:

```
PATCH /restapis/RESTAPI_ID/stages/STAGE_NAME
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "patchOperations": [{
        "op": "replace",
        "path": "/documentationVersion",
        "value": "VERSION_IDENTIFIER"
    }]
}
```

Download a documentation snapshot associated with a stage

After a version of the documentation parts is associated with a stage, you can export the documentation parts together with the API entity definitions, to an external file, using the API Gateway console, the API Gateway REST API, one of its SDKs, or the AWS CLI for API Gateway. The process is the same as for exporting the API. The exported file format can be JSON or YAML.

Using the API Gateway REST API, you can also explicitly set the `extension=documentation,integrations,authorizers` query parameter to include the API documentation parts, API integrations and authorizers in an API export. By default, documentation parts are included, but integrations and authorizers are excluded, when you export an API. The default output from an API export is suited for distribution of the documentation.

To export the API documentation in an external JSON OpenAPI file using the API Gateway REST API, submit the following `GET` request:

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?extensions=documentation
 HTTP/1.1
Accept: application/json
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

Here, the `x-amazon-apigateway-documentation` object contains the documentation parts and the API entity definitions contains the documentation properties supported by OpenAPI. The output does not include details of integration or Lambda authorizers (formerly known as custom authorizers). To include both details, set `extensions=integrations,authorizers,documentation`. To include details of integrations but not of authorizers, set `extensions=integrations,documentation`.

You must set the `Accept:application/json` header in the request to output the result in a JSON file. To produce the YAML output, change the request header to `Accept:application/yaml`.

As an example, we will look at an API that exposes a simple `GET` method on the root resource (`/`). This API has four API entities defined in an OpenAPI definition file, one for each of the `API`, MODEL, METHOD, and RESPONSE types. A documentation part has been added to each of the API, METHOD, and RESPONSE entities. Calling the preceding documentation-exporting command, we get the following output, with the documentation parts listed within the x-amazon-apigateway-documentation object as an extension to a standard OpenAPI file.

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "description": "API info description",
      "version": "2016-11-22T22:39:14Z",
      "title": "doc",
      "x-bar": "API info x-bar"
   },
   "paths": {
      "/": {
         "get": {
            "description": "Method description.",
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               }
            },
            "x-example": "x- Method example"
         },
         "x-bar": "resource x-bar"
      }
   },
   "x-amazon-apigateway-documentation": {
      "version": "1.0.0",
      "createdDate": "2016-11-22T22:41:40Z",
      "documentationParts": [
         {
            "location": {
               "type": "API"
            },
            "properties": {
               "description": "API description",
               "foo": "API foo",
               "x-bar": "API x-bar",
               "info": {
                  "description": "API info description",
                  "version": "API info version",
                  "foo": "API info foo",
                  "x-bar": "API info x-bar"
               }
            }
         },
         {
            "location": {
               "type": "METHOD",
               "method": "GET"
            },
            "properties": {
               "description": "Method description.",
               "x-example": "x- Method example",
               "foo": "Method foo",
               "info": {
                  "version": "method info version",
                  "description": "method info description",
                  "foo": "method info foo"
               }
            }
         },
         {
            "location": {
               "type": "RESOURCE"
            },
            "properties": {
               "description": "resource description",
               "foo": "resource foo",
               "x-bar": "resource x-bar",
               "info": {
                  "description": "resource info description",
                  "version": "resource info version",
                  "foo": "resource info foo",
                  "x-bar": "resource info x-bar"
               }
            }
         }
      ]
   },
   "x-bar": "API x-bar",
   "servers": [
      {
         "url": "https://rznaap68yi.execute-api.ap-southeast-1.amazonaws.com/
{basePath}",
         "variables": {
            "basePath": {
              "default": "/test"
            }
         }
      }
   ],
   "components": {
      "schemas": {
         "Empty": {
            "type": "object",
            "title": "Empty Schema"
         }
      }
   }
}
```

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "description" : "API info description",
    "version" : "2016-11-22T22:39:14Z",
    "title" : "doc",
    "x-bar" : "API info x-bar"
  },
  "host" : "rznaap68yi.execute-api.ap-southeast-1.amazonaws.com",
  "basePath" : "/test",
  "schemes" : [ "https" ],
  "paths" : {
    "/" : {
      "get" : {
        "description" : "Method description.",
        "produces" : [ "application/json" ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "schema" : {
              "$ref" : "#/definitions/Empty"
            }
          }
        },
        "x-example" : "x- Method example"
      },
      "x-bar" : "resource x-bar"
    }
  },
  "definitions" : {
    "Empty" : {
      "type" : "object",
      "title" : "Empty Schema"
    }
  },
  "x-amazon-apigateway-documentation" : {
    "version" : "1.0.0",
    "createdDate" : "2016-11-22T22:41:40Z",
    "documentationParts" : [ {
      "location" : {
        "type" : "API"
      },
      "properties" : {
        "description" : "API description",
        "foo" : "API foo",
        "x-bar" : "API x-bar",
        "info" : {
          "description" : "API info description",
          "version" : "API info version",
          "foo" : "API info foo",
          "x-bar" : "API info x-bar"
        }
      }
    }, {
      "location" : {
        "type" : "METHOD",
        "method" : "GET"
      },
      "properties" : {
        "description" : "Method description.",
        "x-example" : "x- Method example",
        "foo" : "Method foo",
        "info" : {
          "version" : "method info version",
          "description" : "method info description",
          "foo" : "method info foo"
        }
      }
    }, {
      "location" : {
        "type" : "RESOURCE"
      },
      "properties" : {
        "description" : "resource description",
        "foo" : "resource foo",
        "x-bar" : "resource x-bar",
        "info" : {
          "description" : "resource info description",
          "version" : "resource info version",
          "foo" : "resource info foo",
          "x-bar" : "resource info x-bar"
        }
      }
    } ]
  },
  "x-bar" : "API x-bar"
}
```

For an OpenAPI-compliant attribute defined in the `properties` map of a documentation part, API Gateway inserts the attribute into the associated API entity definition. An attribute of `x-`*`something`* is a standard OpenAPI extension. This extension gets propagated into the API entity definition. For example, see the `x-example` attribute for the `GET` method. An attribute like `foo` is not part of the OpenAPI specification and is not injected into its associated API entity definitions.

If a documentation-rendering tool (e.g., OpenAPI UI) parses the API entity definitions to extract documentation attributes, any non OpenAPI-compliant `properties` attributes of a `DocumentationPart`' instance are not available for the tool. However, if a documentation-rendering tool parses the `x-amazon-apigateway-documentation` object to get content, or if the tool calls restapi:documentation-parts and documenationpart:by-id to retrieve documentation parts from API Gateway, all the documentation attributes are available for the tool to display.

To export the documentation with API entity definitions containing integration details to a JSON OpenAPI file, submit the following GET request:

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?
extensions=integrations,documentation HTTP/1.1
Accept: application/json
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

To export the documentation with API entity definitions containing details of integrations and authorizers to a YAML OpenAPI file, submit the following GET request:

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?
extensions=integrations,authorizers,documentation HTTP/1.1
Accept: application/yaml
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

To use the API Gateway console to export and download the published documentation of an API, follow the instructions in Export REST API using the API Gateway console.

Import API documentation

As with importing API entity definitions, you can import documentation parts from an external OpenAPI file into an API in API Gateway. You specify the to-be-imported documentation parts within the x-amazon-apigateway-documentation object extension in a valid OpenAPI definition file. Importing documentation does not alter the existing API entity definitions.

You have an option to merge the newly specified documentation parts into existing documentation parts in API Gateway or to overwrite the existing documentation parts. In the MERGE mode, a new documentation part defined in the OpenAPI file is added to the DocumentationParts collection of the API. If an imported DocumentationPart already exists, an imported attribute replaces the existing one if the two are different. Other existing documentation attributes remain unaffected. In the OVERWRITE mode, the entire DocumentationParts collection is replaced according to the imported OpenAPI definition file.

Importing documentation parts using the API Gateway REST API

To import API documentation using the API Gateway REST API, call the documentationpart:import operation. The following example shows how to overwrite existing documentation parts of an API with a single GET / method, returning a 200 OK response when successful.

OpenAPI 3.0

```
PUT /restapis/<restapi_id>/documentation/parts&mode=overwrite&failonwarnings=true
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
   "openapi": "3.0.0",
   "info": {
      "description": "description",
      "version": "1",
      "title": "doc"
   },
   "paths": {
      "/": {
         "get": {
            "description": "Method description.",
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               }
            }
         }
      }
   },
   "x-amazon-apigateway-documentation": {
      "version": "1.0.3",
      "documentationParts": [
         {
            "location": {
               "type": "API"
            },
            "properties": {
               "description": "API description",
               "info": {
                  "description": "API info description 4",
                  "version": "API info version 3"
               }
            }
         },
         {
            "location": {
               "type": "METHOD",
               "method": "GET"
            },
            "properties": {
               "description": "Method description."
            }
         },
         {
            "location": {
               "type": "MODEL",
               "name": "Empty"
            },
            "properties": {
               "title": "Empty Schema"
            }
         },
         {
            "location": {
               "type": "RESPONSE",
               "method": "GET",
               "statusCode": "200"
            },
            "properties": {
               "description": "200 response"
```

```
                }
            }
        ]
    },
    "servers": [
        {
            "url": "/"
        }
    ],
    "components": {
        "schemas": {
            "Empty": {
                "type": "object",
                "title": "Empty Schema"
            }
        }
    }
}
```

OpenAPI 2.0

```
PUT /restapis/<restapi_id>/documentation/parts&mode=overwrite&failonwarnings=true
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "swagger": "2.0",
  "info": {
    "description": "description",
    "version": "1",
    "title": "doc"
  },
  "host": "",
  "basePath": "/",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "description": "Method description.",
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  },
  "x-amazon-apigateway-documentation": {
    "version": "1.0.3",
    "documentationParts": [
      {
        "location": {
          "type": "API"
        },
        "properties": {
          "description": "API description",
          "info": {
            "description": "API info description 4",
            "version": "API info version 3"
          }
        }
      },
      {
        "location": {
          "type": "METHOD",
          "method": "GET"
        },
        "properties": {
          "description": "Method description."
        }
      },
      {
        "location": {
          "type": "MODEL",
          "name": "Empty"
        },
        "properties": {
          "title": "Empty Schema"
        }
      },
      {
        "location": {
          "type": "RESPONSE",
          "method": "GET",
          "statusCode": "200"
        },
        "properties": {
          "description": "200 response"
        }
      }
    ]
  }
}
```

When successful, this request returns a 200 OK response containing the imported `DocumentationPartId` in the payload.

```
{
  "ids": [
    "kg3mth",
    "796rtf",
    "zhek4p",
    "5ukm9s"
  ]
}
```

In addition, you can also call restapi:import or restapi:put, supplying the documentation parts in the `x-amazon-apigateway-documentation` object as part of the input OpenAPI file of the API definition. To exclude the documentation parts from the API import, set `ignore=documentation` in the request query parameters.

Importing documentation parts using the API Gateway console

The following instructions describe how to import documentation parts.

To use the console to import documentation parts of an API from an external file

1. In the main navigation pane, choose **Documentation**.
2. Choose **Import**.
3. If you have existing documentation, select to either **Overwrite** or **Merge** your new documentation.
4. Choose **Choose file** to load a file from a drive, or enter file contents into the file view. For an example, see the payload of the example request in Importing documentation parts using the API Gateway REST API.
5. Choose how to handle warnings on import. Select either **Fail on warnings** or **Ignore warnings**. For more information, see the section called "Errors and warnings during import".
6. Choose **Import**.

Control access to API documentation

If you have a dedicated documentation team to write and edit your API documentation, you can configure separate access permissions for your developers (for API development) and for your writers or editors (for content development). This is especially appropriate when a third-party vendor is involved in creating the documentation for you.

To grant your documentation team the access to create, update, and publish your API documentation, you can assign the documentation team an IAM role with the following IAM policy, where *account_id* is the AWS account ID of your documentation team.

```
{
  "Version": "2012-10-17",
  "Statement": [

    {
      "Sid": "StmtDocPartsAddEditViewDelete",
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway::account_id:/restapis/*/documentation/*"
      ]
    }
  ]
}
```

For information on setting permissions to access API Gateway resources, see the section called “How Amazon API Gateway works with IAM”.

Generating an SDK for a REST API in API Gateway

To call your REST API in a platform- or language-specific way, you must generate the platform- or language-specific SDK of the API. Currently, API Gateway supports generating an SDK for an API in Java, JavaScript, Java for Android, Objective-C or Swift for iOS, and Ruby.

This section explains how to generate an SDK of an API Gateway API. It also demonstrates how to use the generated SDK in a Java app, a Java for Android app, Objective-C and Swift for iOS apps, and a JavaScript app.

To facilitate the discussion, we use this API Gateway API, which exposes this Simple Calculator Lambda function.

Before proceeding, create or import the API and deploy it at least once in API Gateway. For instructions, see Deploying a REST API in Amazon API Gateway.

Topics

Generate SDKs for an API using the API Gateway console

To generate a platform- or language-specific SDK for an API in API Gateway, you must first create, test, and deploy the API in a stage. For illustration purposes, we use the Simple Calculator API as an example to generate language-specific or platform-specific SDKs throughout this section. For instructions on how to create, test, and deploy this API, see Create the Simple Calculator API.

Topics

- Generate the Java SDK of an API
- Generate the Android SDK of an API
- Generate the iOS SDK of an API
- Generate the JavaScript SDK of a REST API
- Generate the Ruby SDK of an API

Generate the Java SDK of an API

To generate the Java SDK of an API in API Gateway

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Choose **Stages**.
4. In the **Stages** pane, select the name of the stage.
5. Open the **Stage actions** menu, and then choose **Generate SDK**.
6. For **Platform**, choose the **Java** platform and do the following:

   a. For **Service Name**, specify the name of your SDK. For example, **`SimpleCalcSdk`**. This becomes the name of your SDK client class. The name corresponds to the `<name>` tag under `<project>` in the pom.xml file, which is in the SDK's project folder. Do not include hyphens.

   b. For **Java Package Name**, specify a package name for your SDK. For example, **`examples.aws.apig.simpleCalc.sdk`**. This package name is used as the namespace of your SDK library. Do not include hyphens.

   c. For **Java Build System**, enter **`maven`** or **`gradle`** to specify the build system.

d. For **Java Group Id**, enter a group identifier for your SDK project. For example, enter **`my-apig-api-examples`**. This identifier corresponds to the `<groupId>` tag under `<project>` in the `pom.xml` file, which is in the SDK's project folder.

e. For **Java Artifact Id**, enter an artifact identifier for your SDK project. For example, enter **`simple-calc-sdk`**. This identifier corresponds to the `<artifactId>` tag under `<project>` in the `pom.xml` file, which is in the SDK's project folder.

f. For **Java Artifact Version**, enter a version identifier string. For example, **`1.0.0`**. This version identifier corresponds to the `<version>` tag under `<project>` in the `pom.xml` file, which is in the SDK's project folder.

g. For **Source Code License Text**, enter the license text of your source code, if any.

7. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.

Follow the instructions in Use a Java SDK generated by API Gateway for a REST API to use the generated SDK.

Every time you update an API, you must redeploy the API and regenerate the SDK to have the updates included.

Generate the Android SDK of an API

To generate the Android SDK of an API in API Gateway

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Choose **Stages**.
4. In the **Stages** pane, select the name of the stage.
5. Open the **Stage actions** menu, and then choose **Generate SDK**.
6. For **Platform**, choose the Android platform and do the following:

   a. For **Group ID**, enter the unique identifier for the corresponding project. This is used in the `pom.xml` file (for example, **`com.mycompany`**).

   b. For **Invoker package**, enter the namespace for the generated client classes (for example, **`com.mycompany.clientsdk`**).

   c. For **Artifact ID**, enter the name of the compiled .jar file without the version. This is used in the `pom.xml` file (for example, **`aws-apigateway-api-sdk`**).

   d. For **Artifact version**, enter the artifact version number for the generated client. This is used in the `pom.xml` file and should follow a *`major`*.*`minor`*.*`patch`* pattern (for example, **`1.0.0`**).

7. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.

Follow the instructions in Use an Android SDK generated by API Gateway for a REST API to use the generated SDK.

Every time you update an API, you must redeploy the API and regenerate the SDK to have the updates included.

Generate the iOS SDK of an API

To generate the iOS SDK of an API in API Gateway

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Choose **Stages**.
4. In the **Stages** pane, select the name of the stage.
5. Open the **Stage actions** menu, and then choose **Generate SDK**.
6. For **Platform**, choose the **iOS (Objective-C) or iOS (Swift)** platform and do the following:

   - Type a unique prefix in the **Prefix** box.

     The effect of prefix is as follows: if you assign, for example, **`SIMPLE_CALC`** as the prefix for the SDK of the SimpleCalc API with `input`, `output`, and `result` models, the generated SDK will contain the `SIMPLE_CALCSimpleCalcClient` class that encapsulates the API, including the method requests/responses. In addition, the generated SDK will contain the `SIMPLE_CALCinput`, `SIMPLE_CALCoutput`, and `SIMPLE_CALCresult` classes to represent the input, output, and results, respectively, to represent the request input and response output. For more information, see Use iOS SDK generated by API Gateway for a REST API in Objective-C or Swift.

7. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.

Follow the instructions in Use iOS SDK generated by API Gateway for a REST API in Objective-C or Swift to use the generated SDK.

Every time you update an API, you must redeploy the API and regenerate the SDK to have the updates included.

Generate the JavaScript SDK of a REST API

To generate the JavaScript SDK of an API in API Gateway

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Choose **Stages**.
4. In the **Stages** pane, select the name of the stage.
5. Open the **Stage actions** menu, and then choose **Generate SDK**.
6. For **Platform**, choose the **JavaScript** platform.
7. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.

Follow the instructions in Use a JavaScript SDK generated by API Gateway for a REST API to use the generated SDK.

Every time you update an API, you must redeploy the API and regenerate the SDK to have the updates included.

Generate the Ruby SDK of an API

To generate the Ruby SDK of an API in API Gateway

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a REST API.
3. Choose **Stages**.
4. In the **Stages** pane, select the name of the stage.
5. Open the **Stage actions** menu, and then choose **Generate SDK**.
6. For **Platform**, choose the **Ruby** platform and do the following:

   a. For **Service Name**, specify the name of your SDK. For example, **`SimpleCalc`**. This is used to generate the Ruby Gem namespace of your API. The name must be all letters, (`a-zA-Z`), without any other special characters or numbers.

   b. For **Ruby Gem Name**, specify the name of the Ruby Gem to contain the generated SDK source code for your API. By default, it is the lower-cased service name plus the `-sdk` suffix—for example, **`simplecalc-sdk`**.

   c. For **Ruby Gem Version**, specify a version number for the generated Ruby Gem. By default, it is set to `1.0.0`.

7. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.

Follow the instructions in Use a Ruby SDK generated by API Gateway for a REST API to use the generated SDK.

Every time you update an API, you must redeploy the API and regenerate the SDK to have the updates included.

Generate SDKs for an API using AWS CLI commands

You can use AWS CLI to generate and download an SDK of an API for a supported platform by calling the get-sdk command. We demonstrate this for some of the supported platforms in the following.

Topics

- Generate and download the Java for Android SDK using the AWS CLI
- Generate and download the JavaScript SDK using the AWS CLI
- Generate and download the Ruby SDK using the AWS CLI

Generate and download the Java for Android SDK using the AWS CLI

To generate and download a Java for Android SDK generated by API Gateway of an API (`udpuvvzbkc`) at a given stage (`test`), call the command as follows:

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test \
            --sdk-type android \
            --parameters groupId='com.mycompany',\
                invokerPackage='com.mycompany.myApiSdk',\
                artifactId='myApiSdk',\
                artifactVersion='0.0.1' \
            ~/apps/myApi/myApi-android-sdk.zip
```

The last input of `~/apps/myApi/myApi-android-sdk.zip` is the path to the downloaded SDK file named `myApi-android-sdk.zip`.

Generate and download the JavaScript SDK using the AWS CLI

To generate and download a JavaScript SDK generated by API Gateway of an API (`udpuvvzbkc`) at a given stage (`test`), call the command as follows:

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test \
            --sdk-type javascript \
            ~/apps/myApi/myApi-js-sdk.zip
```

The last input of `~/apps/myApi/myApi-js-sdk.zip` is the path to the downloaded SDK file named `myApi-js-sdk.zip`.

Generate and download the Ruby SDK using the AWS CLI

To generate and download a Ruby SDK of an API (`udpuvvzbkc`) at a given stage (`test`), call the command as follows:

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test  \
            --sdk-type ruby \
            --parameters service.name=myApiRubySdk,ruby.gem-name=myApi,ruby.gem-
version=0.01 \
            ~/apps/myApi/myApi-ruby-sdk.zip
```

The last input of `~/apps/myApi/myApi-ruby-sdk.zip` is the path to the downloaded SDK file named `myApi-ruby-sdk.zip`.

Next, we show how to use the generated SDK to call the underlying API. For more information, see Call REST API through generated SDKs.

Simple calculator Lambda function

As an illustration, we will use a Node.js Lambda function that performs the binary operations of addition, subtraction, multiplication and division.

Topics

- Simple calculator Lambda function input format
- Simple calculator Lambda function output format
- Simple calculator Lambda function implementation

Simple calculator Lambda function input format

This function takes an input of the following format:

```
{ "a": "Number", "b": "Number", "op": "string"}
```

where op can be any of `(+, -, *, /, add, sub, mul, div)`.

Simple calculator Lambda function output format

When an operation succeeds, it returns the result of the following format:

```
{ "a": "Number", "b": "Number", "op": "string", "c": "Number"}
```

where `c` holds the result of the calculation.

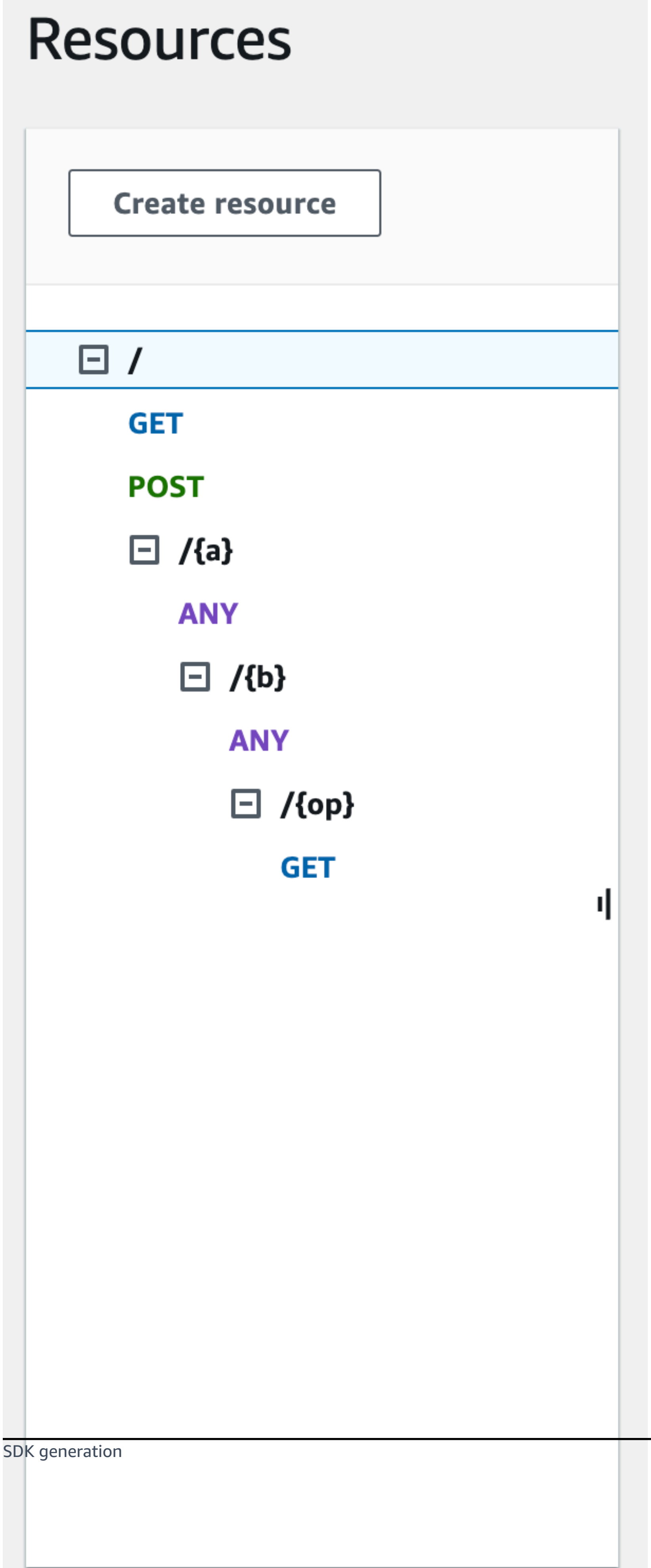
Simple calculator Lambda function implementation

The implementation of the Lambda function is as follows:

```
export const handler = async function (event, context) {
  console.log("Received event:", JSON.stringify(event));

  if (
    event.a === undefined ||
    event.b === undefined ||
    event.op === undefined
  ) {
    return "400 Invalid Input";
  }

  const res = {};
  res.a = Number(event.a);
  res.b = Number(event.b);
  res.op = event.op;
  if (isNaN(event.a) || isNaN(event.b)) {
    return "400 Invalid Operand";
  }
  switch (event.op) {
    case "+":
    case "add":
      res.c = res.a + res.b;
      break;
    case "-":
    case "sub":
      res.c = res.a - res.b;
      break;
    case "*":
    case "mul":
      res.c = res.a * res.b;
      break;
    case "/":
    case "div":
      if (res.b == 0) {
        return "400 Divide by Zero";
      } else {
        res.c = res.a / res.b;
      }
      break;
    default:
      return "400 Invalid Operator";
  }

  return res;
};
```

Simple calculator API in API Gateway

Our simple calculator API exposes three methods (GET, POST, GET) to invoke the the section called "Simple calculator Lambda function". A graphical representation of this API is shown as follows:


Resources
Create resource
/
GET
POST
/{a}
ANY
/{b}
ANY
/{op}
GET

These three methods show different ways to supply the input for the backend Lambda function to perform the same operation:

- The `GET /?a=...&b=...&op=...` method uses the query parameters to specify the input.
- The `POST /` method uses a JSON payload of `{"a":"Number", "b":"Number", "op":"string"}` to specify the input.
- The `GET /{a}/{b}/{op}` method uses the path parameters to specify the input.

If not defined, API Gateway generates the corresponding SDK method name by combining the HTTP method and path parts. The root path part (`/`) is referred to as `Api Root`. For example, the default Java SDK method name for the API method of `GET /?a=...&b=...&op=...` is `getABOp`, the default SDK method name for `POST /` is `postApiRoot`, and the default SDK method name for `GET /{a}/{b}/{op}` is `getABOp`. Individual SDKs may customize the convention. Consult the documentation in the generated SDK source for SDK specific method names.

You can, and should, override the default SDK method names by specifying the operationName property on each API method. You can do so when creating the API method or updating the API method using the API Gateway REST API. In the API Swagger definition, you can set the `operationId` to achieve the same result.

Before showing how to call these methods using an SDK generated by API Gateway for this API, let's recall briefly how to set them up. For detailed instructions, see Creating a REST API in Amazon API Gateway. If you're new to API Gateway, see Build an API Gateway REST API with Lambda integration first.

Create models for input and output

To specify strongly typed input in the SDK, we create an `Input` model for the API. To describe the response body data type, we create an `Output` model and a `Result` model.

To create models for the input, output, and result

1. In the main navigation pane, choose **Models**.
2. Choose **Create model**.
3. For **Name**, enter **`input`**.
4. For **Content type**, enter **`application/json`**.

If no matching content type is found, request validation is not performed. To use the same model regardless of the content type, enter **$default**.

5. For **Model schema**, enter the following model:

```
{
    "$schema" : "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"object",
    "properties":{
        "a":{"type":"number"},
        "b":{"type":"number"},
        "op":{"type":"string"}
    },
    "title":"Input"
}
```

6. Choose **Create model**.

7. Repeat the following steps to create an `Output` model and a `Result` model.

For the `Output` model, enter the following for the **Model schema**:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "c": {"type":"number"}
    },
    "title": "Output"
}
```

For the `Result` model, enter the following for the **Model schema**. Replace the API ID `abc123` with your API ID.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"object",
    "properties":{
        "input":{
            "$ref":"https://apigateway.amazonaws.com/restapis/abc123/models/Input"
        },
        "output":{
            "$ref":"https://apigateway.amazonaws.com/restapis/abc123/models/Output"
        }
    },
    "title":"Result"
}
```

Set up GET / method query parameters

For the `GET /?a=..&b=..&op=..` method, the query parameters are declared in **Method Request**:

To set up GET / URL query string parameters

1. In the **Method request** section for the `GET` method on the root (/) resource, choose **Edit**.
2. Choose **URL query string parameters** and do the following:

    a. Choose **Add query string**.

    b. For **Name**, enter **a**.

    c. Keep **Required** and **Caching** turned off.

    d. Keep **Caching** turned off.

    Repeat the same steps and create a query string named **b** and a query string named **op**.
3. Choose **Save**.

Set up data model for the payload as input to the backend

For the `POST /` method, we create the `Input` model and add it to the method request to define the shape of input data.

To set up the data model for the payload as input to the backend

1. In the **Method request** section, for the `POST` method on the root (/) resource choose **Edit**.
2. Choose **Request body**.
3. Choose **Add model**.
4. For **Content type**, enter **`application/json`**.
5. For **Model**, select **Input**.

6. Choose **Save**.

With this model, your API customers can call the SDK to specify the input by instantiating an `Input` object. Without this model, your customers would be required to create dictionary object to represent the JSON input to the Lambda function.

Set up data model for the result output from the backend

For all three methods, we create the `Result` model and add it to the method's `Method Response` to define the shape of output returned by the Lambda function.

To set up the data model for the result output from the backend

1. Select the **/{a}/{b}/{op}** resource, and then choose the **GET** method.
2. On the **Method response** tab, under **Response 200**, choose **Edit**.
3. Under **Response body**, choose **Add model**.
4. For **Content type**, enter **`application/json`**.
5. For **Model**, select **Result**.
6. Choose **Save**.

With this model, your API customers can parse a successful output by reading properties of a `Result` object. Without this model, customers would be required to create dictionary object to represent the JSON output.

Simple calculator API OpenAPI definition

The following is the OpenAPI definition of the simple calculator API. You can import it into your account. However, you need to reset the resource-based permissions on the [Lambda function](#) after the import. To do so, re-select the Lambda function that you created in your account from the **Integration Request** in the API Gateway console. This will cause the API Gateway console to reset the required permissions. Alternatively, you can use AWS Command Line Interface for Lambda command of [add-permission](#).

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-29T20:27:30Z",
    "title": "SimpleCalc"
  },
  "host": "t6dve4zn25.execute-api.us-west-2.amazonaws.com",
  "basePath": "/demo",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "op",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "a",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "b",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "Input",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Input"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      }
    },
    "/{a}": {
      "x-amazon-apigateway-any-method": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "404": {
            "description": "404 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match",
          "responses": {
            "default": {
              "statusCode": "404",
              "responseTemplates": {
                "application/json": "{ \"Message\" : \"Can't $context.httpMethod
 $context.resourcePath\" }"
              }
            }
          },
          "type": "mock"
        }
      }
    },
    "/{a}/{b}": {
      "x-amazon-apigateway-any-method": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "b",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "404": {
            "description": "404 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match",
          "responses": {
            "default": {
              "statusCode": "404",
              "responseTemplates": {
                "application/json": "{ \"Message\" : \"Can't $context.httpMethod
 $context.resourcePath\" }"
              }
            }
          },
          "type": "mock"
        }
      }
    },
    "/{a}/{b}/{op}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "b",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "op",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
    "Input": {
      "type": "object",
      "properties": {
        "a": {
          "type": "number"
        },
        "b": {
          "type": "number"
        },
        "op": {
          "type": "string"
        }
      },
      "title": "Input"
    },
    "Output": {
      "type": "object",
      "properties": {
        "c": {
          "type": "number"
        }
      },
      "title": "Output"
    },
    "Result": {
      "type": "object",
      "properties": {
        "input": {
          "$ref": "#/definitions/Input"
        },
        "output": {
          "$ref": "#/definitions/Output"
        }
      },
      "title": "Result"
    }
  }
}
```

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "SimpleCalc",
    "version" : "2016-09-29T20:27:30Z"
  },
  "servers" : [ {
    "url" : "https://t6dve4zn25.execute-api.us-west-2.amazonaws.com/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "demo"
      }
    }
  } ],
  "paths" : {
    "/{a}/{b}" : {
      "x-amazon-apigateway-any-method" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "b",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "404" : {
            "description" : "404 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "mock",
          "responses" : {
            "default" : {
              "statusCode" : "404",
              "responseTemplates" : {
                "application/json" : "{ \"Message\" : \"Can't $context.httpMethod 
 $context.resourcePath\" }"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "{\"statusCode\": 200}"
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    },
    "/{a}/{b}/{op}" : {
      "get" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "b",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "op",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Result"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "passthroughBehavior" : "when_no_templates"
        }
      }
    },
    "/" : {
      "get" : {
        "parameters" : [ {
          "name" : "op",
          "in" : "query",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "a",
          "in" : "query",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "b",
          "in" : "query",
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Result"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "passthroughBehavior" : "when_no_templates"
        }
      },
      "post" : {
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/Input"
              }
            }
          },
          "required" : true
        },
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Result"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    },
    "/{a}" : {
      "x-amazon-apigateway-any-method" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "404" : {
            "description" : "404 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "mock",
          "responses" : {
            "default" : {
              "statusCode" : "404",
              "responseTemplates" : {
                "application/json" : "{ \"Message\" : \"Can't $context.httpMethod 
 $context.resourcePath\" }"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "{\"statusCode\": 200}"
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "Input" : {
        "title" : "Input",
        "type" : "object",
        "properties" : {
          "a" : {
            "type" : "number"
          },
          "b" : {
            "type" : "number"
          },
          "op" : {
            "type" : "string"
          }
        }
      },
      "Output" : {
        "title" : "Output",
        "type" : "object",
        "properties" : {
          "c" : {
              "type" : "number"
            }
          }
        },
        "Result" : {
          "title" : "Result",
          "type" : "object",
          "properties" : {
            "input" : {
              "$ref" : "#/components/schemas/Input"
            },
            "output" : {
              "$ref" : "#/components/schemas/Output"
            }
          }
        }
      }
    }
}
```

Sell your API Gateway APIs through AWS Marketplace

After you build, test, and deploy your APIs, you can package them in an API Gateway usage plan and sell the plan as a Software as a Service (SaaS) product through AWS Marketplace. API buyers subscribing to your product offering are billed by AWS Marketplace based on the number of requests made to the usage plan.

To sell your APIs on AWS Marketplace, you must set up the sales channel to integrate AWS Marketplace with API Gateway. Generally speaking, this involves listing your product on AWS Marketplace, setting up an IAM role with appropriate policies to allow API Gateway to send usage metrics to AWS Marketplace, associating an AWS Marketplace product with an API Gateway usage plan, and associating an AWS Marketplace buyer with an API Gateway API key. Details are discussed in the following sections.

For more information about selling your API as a SaaS product on AWS Marketplace, see the AWS Marketplace User Guide.

Topics

Initialize AWS Marketplace integration with API Gateway

The following tasks are for one-time initialization of AWS Marketplace integration with API Gateway, which enables you to sell your APIs as a SaaS product.

List a product on AWS Marketplace

To list your usage plan as a SaaS product, submit a product load form through AWS Marketplace. The product must contain a dimension named `apigateway` of the `requests` type. This dimension defines the price-per-request and is used by API Gateway to meter requests to your APIs.

Create the metering role

Create an IAM role named `ApiGatewayMarketplaceMeteringRole` with the following execution policy and trust policy. This role allows API Gateway to send usage metrics to AWS Marketplace on your behalf.

Execution policy of the metering role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "aws-marketplace:BatchMeterUsage",
        "aws-marketplace:ResolveCustomer"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

Trusted relationship policy of the metering role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Associate usage plan with AWS Marketplace product

When you list a product on AWS Marketplace, you receive an AWS Marketplace product code. To integrate API Gateway with AWS Marketplace, associate your usage plan with the AWS Marketplace product code. You enable the association by setting the API Gateway `UsagePlan`'s [productCode] field to your AWS Marketplace product code, using the API Gateway console, the API Gateway REST API, the AWS CLI for API Gateway, or AWS SDK for API Gateway. The following code example uses the API Gateway REST API:

```
PATCH /usageplans/USAGE_PLAN_ID
Host: apigateway.region.amazonaws.com
Authorization: ...

{
    "patchOperations" : [{
        "path" : "/productCode",
        "value" : "MARKETPLACE_PRODUCT_CODE",
        "op" : "replace"
    }]
}
```

Handle customer subscription to usage plans

The following tasks are handled by your developer portal application.

When a customer subscribes to your product through AWS Marketplace, AWS Marketplace forwards a `POST` request to the SaaS subscriptions URL that you registered when listing your product on AWS Marketplace. The `POST` request comes with an `x-amzn-marketplace-token` parameter containing buyer information. Follow the instructions in SaaS customer onboarding to handle this redirect in your developer portal application.

Responding to a customer's subscribing request, AWS Marketplace sends a `subscribe-success` notification to an Amazon SNS topic that you can subscribe to. (See SaaS customer onboarding). To accept the customer subscription request, you handle the `subscribe-success` notification by creating or retrieving an API Gateway API key for the customer, associating the customer's AWS

Marketplace-provisioned `customerId` with the API keys, and then adding the API key to your usage plan.

When the customer's subscription request completes, the developer portal application should present the customer with the associated API key and inform the customer that the API key must be included in the `x-api-key` header in requests to the APIs.

When a customer cancels a subscription to a usage plan, AWS Marketplace sends an `unsubscribe-success` notification to the SNS topic. To complete the process of unsubscribing the customer, you handle the `unsubscribe-success` notification by removing the customer's API keys from the usage plan.

Authorize a customer to access a usage plan

To authorize access to your usage plan for a given customer, use the API Gateway API to fetch or create an API key for the customer and add the API key to the usage plan.

The following example shows how to call the API Gateway REST API to create a new API key with a specific AWS Marketplace `customerId` value (*MARKETPLACE_CUSTOMER_ID*).

```
POST apikeys HTTP/1.1
Host: apigateway.region.amazonaws.com
Authorization: ...

{
  "name" : "my_api_key",
  "description" : "My API key",
  "enabled" : "false",
  "stageKeys" : [ {
    "restApiId" : "uycll6xg9a",
    "stageName" : "prod"
  } ],
  "customerId" : "MARKETPLACE_CUSTOMER_ID"
}
```

The following example shows how to get an API key with a specific AWS Marketplace `customerId` value (*MARKETPLACE_CUSTOMER_ID*).

```
GET apikeys?customerId=MARKETPLACE_CUSTOMER_ID HTTP/1.1
Host: apigateway.region.amazonaws.com
Authorization: ...
```

To add an API key to a usage plan, create a UsagePlanKey with the API key for the relevant usage plan. The following example shows how to accomplish this using the API Gateway REST API, where n371pt is the usage plan ID and q5ugs7qjjh is an example API keyId returned from the preceding examples.

```
POST /usageplans/n371pt/keys HTTP/1.1
Host: apigateway.region.amazonaws.com
Authorization: ...

{
    "keyId": "q5ugs7qjjh",
    "keyType": "API_KEY"
}
```

Associate a customer with an API key

You must update the ApiKey's customerId field to the AWS Marketplace customer ID of the customer. This associates the API key with the AWS Marketplace customer, which enables metering and billing for the buyer. The following code example calls the API Gateway REST API to do that.

```
PATCH /apikeys/q5ugs7qjjh
Host: apigateway.region.amazonaws.com
Authorization: ...

{
    "patchOperations" : [{
        "path" : "/customerId",
        "value" : "MARKETPLACE_CUSTOMER_ID",
        "op" : "replace"
    }]
}
```

Protecting your REST API

API Gateway provides a number of ways to protect your API from certain threats, like malicious users or spikes in traffic. You can protect your API using strategies like generating SSL certificates, configuring a web application firewall, setting throttling targets, and only allowing access to your API from a Virtual Private Cloud (VPC). In this section you can learn how to enable these capabilities using API Gateway.

Topics

Configuring mutual TLS authentication for a REST API

Mutual TLS authentication requires two-way authentication between the client and the server. With mutual TLS, clients must present X.509 certificates to verify their identity to access your API. Mutual TLS is a common requirement for Internet of Things (IoT) and business-to-business applications.

You can use mutual TLS along with other authorization and authentication operations that API Gateway supports. API Gateway forwards the certificates that clients provide to Lambda authorizers and to backend integrations.

> ⚠ **Important**
>
> By default, clients can invoke your API by using the `execute-api` endpoint that API Gateway generates for your API. To ensure that clients can access your API only by using a custom domain name with mutual TLS, disable the default `execute-api` endpoint. To learn more, see the section called “Disable the default endpoint”.

Topics

- Troubleshooting domain name status messages

Prerequisites for mutual TLS

To configure mutual TLS you need:

- A custom domain name
- At least one certificate configured in AWS Certificate Manager for your custom domain name
- A truststore configured and uploaded to Amazon S3

Custom domain names

To enable mutual TLS for a REST API, you must configure a custom domain name for your API. You can enable mutual TLS for a custom domain name, and then provide the custom domain name to clients. To access an API by using a custom domain name that has mutual TLS enabled, clients must present certificates that you trust in API requests. You can find more information at the section called “Custom domain names”.

Using AWS Certificate Manager issued certificates

You can request a publicly trusted certificate directly from ACM or import public or self-signed certificates. To setup a certificate in ACM, go to ACM. If you would like to import a certificate, continue reading in the following section.

Using an imported or AWS Private Certificate Authority certificate

To use a certificate imported into ACM or a certificate from AWS Private Certificate Authority with mutual TLS, API Gateway needs an `ownershipVerificationCertificate` issued by ACM. This ownership certificate is only used to verify that you have permissions to use the domain name. It is not used for the TLS handshake. If you don't already have a `ownershipVerificationCertificate`, go to https://console.aws.amazon.com/acm/ to set one up.

You will need to keep this certificate valid for the lifetime of your domain name. If a certificate expires and auto-renew fails, all updates to the domain name will be locked. You will need to update the `ownershipVerificationCertificateArn` with a valid `ownershipVerificationCertificate` before you can make any other changes. The `ownershipVerificationCertificate` cannot be used as a server certificate for another mutual TLS domain in API Gateway. If a certificate is directly re-imported into ACM, the issuer must stay the same.

Configuring your truststore

Truststores are text files with a `.pem` file extension. They are a trusted list of certificates from Certificate Authorities. To use mutual TLS, create a truststore of X.509 certificates that you trust to access your API.

You must include the complete chain of trust, starting from the issuing CA certificate, up to the root CA certificate, in your truststore. API Gateway accepts client certificates issued by any CA present in the chain of trust. The certificates can be from public or private certificate authorities. Certificates can have a maximum chain length of four. You can also provide self-signed certificates. The following algorithms are supported in the truststore:

- SHA-256 or stronger
- RSA-2048 or stronger
- ECDSA-256 or stronger

API Gateway validates a number of certificate properties. You can use Lambda authorizers to perform additional checks when a client invokes an API, including checking whether a certificate has been revoked. API Gateway validates the following properties:

Validation	Description
X.509 syntax	The certificate must meet X.509 syntax requirements.
Integrity	The certificate's content must not have been altered from that signed by the certificate authority from the truststore.
Validity	The certificate's validity period must be current.
Name chaining / key chaining	The names and subjects of certificates must form an unbroken chain. Certificates can have a maximum chain length of four.

Upload the truststore to an Amazon S3 bucket in a single file

The following is an example of what a .pem file might look like.

Example certificates.pem

```
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
...
```

The following AWS CLI command uploads `certificates.pem` to your Amazon S3 bucket.

```
aws s3 cp certificates.pem s3://bucket-name
```

Your Amazon S3 bucket must have read permission for API Gateway to allow API Gateway to access your truststore.

Configuring mutual TLS for a custom domain name

To configure mutual TLS for a REST API, you must use a Regional custom domain name for your API, with a TLS_1_2 security policy. For more information about choosing a security policy, see the section called “Choosing a security policy”.

> **Note**
>
> Mutual TLS isn't supported for private APIs.

After you've uploaded your truststore to Amazon S3, you can configure your custom domain name to use mutual TLS. Paste the following (slashes included) into a terminal:

```
aws apigateway create-domain-name --region us-east-2 \
    --domain-name api.example.com \
    --regional-certificate-arn arn:aws:acm:us-
east-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --endpoint-configuration types=REGIONAL \
    --security-policy TLS_1_2 \
    --mutual-tls-authentication truststoreUri=s3://bucket-name/key-name
```

After you create the domain name, you must configure DNS records and basepath mappings for API operations. To learn more, see Setting up a regional custom domain name in API Gateway.

Invoke an API by using a custom domain name that requires mutual TLS

To invoke an API with mutual TLS enabled, clients must present a trusted certificate in the API request. When a client attempts to invoke your API, API Gateway looks for the client certificate's issuer in your truststore. For API Gateway to proceed with the request, the certificate's issuer and the complete chain of trust up to the root CA certificate must be in your truststore.

The following example `curl` command sends a request to `api.example.com,` that includes `my-cert.pem` in the request. `my-key.key` is the private key for the certificate.

```
curl -v --key ./my-key.key --cert ./my-cert.pem api.example.com
```

Your API is invoked only if your truststore trusts the certificate. The following conditions will cause API Gateway to fail the TLS handshake and deny the request with a `403` status code. If your certificate:

- isn't trusted
- is expired
- doesn't use a supported algorithm

> **Note**
>
> API Gateway doesn't verify if a certificate has been revoked.

Updating your truststore

To update the certificates in your truststore, upload a new certificate bundle to Amazon S3. Then, you can update your custom domain name to use the updated certificate.

Use Amazon S3 versioning to maintain multiple versions of your truststore. When you update your custom domain name to use a new truststore version, API Gateway returns warnings if certificates are invalid.

API Gateway produces certificate warnings only when you update your domain name. API Gateway doesn’t notify you if a previously uploaded certificate expires.

The following AWS CLI command updates a custom domain name to use a new truststore version.

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations op='replace',path='/mutualTlsAuthentication/
truststoreVersion',value='abcdef123'
```

Disable mutual TLS

To disable mutual TLS for a custom domain name, remove the truststore from your custom domain name, as shown in the following command.

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations op='replace',path='/mutualTlsAuthentication/
truststoreUri',value=''
```

Troubleshooting certificate warnings

When creating a custom domain name with mutual TLS, API Gateway returns warnings if certificates in the truststore are not valid. This can also occur when updating a custom domain name to use a new truststore. The warnings indicate the issue with the certificate and the subject of the certificate that produced the warning. Mutual TLS is still enabled for your API, but some clients might not be able to access your API.

You'll need to decode the certificates in your truststore in order to identify which certificate produced the warning. You can use tools such as `openssl` to decode the certificates and identify their subjects.

The following command displays the contents of a certificate, including its subject:

```
openssl x509 -in certificate.crt -text -noout
```

Update or remove the certificates that produced warnings, and then upload a new truststore to Amazon S3. After uploading the new truststore, update your custom domain name to use the new truststore.

Troubleshooting domain name conflicts

The error "`The certificate subject <certSubject> conflicts with an existing certificate from a different issuer.`" means multiple Certificate Authorities have issued a certificate for this domain. For each subject in the certificate, there can only be one issuer in API Gateway for mutual TLS domains. You will need to get all of your certificates for that subject through a single issuer. If the problem is with a certificate you don't have control of but you can prove ownership of the domain name, contact AWS Support to open a ticket.

Troubleshooting domain name status messages

`PENDING_CERTIFICATE_REIMPORT`: This means you reimported a certificate to ACM and it failed validation because the new certificate has a SAN (subject alternative name) that is not covered by the `ownershipVerificationCertificate` or the subject or SANs in the certificate don't cover the domain name. Something might be configured incorrectly or an invalid certificate was imported. You need to reimport a valid certificate into ACM. For more information about validation see Validating domain ownership.

`PENDING_OWNERSHIP_VERIFICATION`: This means your previously verified certificate has expired and ACM was unable to auto-renew it. You will need to renew the certificate or request a new certificate. More information about certificate renewal can be found at ACM's troubleshooting managed certificate renewal guide.

Generate and configure an SSL certificate for backend authentication

You can use API Gateway to generate an SSL certificate and then use its public key in the backend to verify that HTTP requests to your backend system are from API Gateway. This allows your HTTP backend to control and accept only requests that originate from Amazon API Gateway, even if the backend is publicly accessible.

> **Note**
>
> Some backend servers might not support SSL client authentication as API Gateway does and could return an SSL certificate error. For a list of incompatible backend servers, see the section called “Important notes”.

The SSL certificates that are generated by API Gateway are self-signed, and only the public key of a certificate is visible in the API Gateway console or through the APIs.

Topics

Generate a client certificate using the API Gateway console

1. Open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Choose a REST API.
3. In the main navigation pane, choose **Client certificates**.
4. From the **Client certificates** page, choose **Generate certificate**.
5. (Optional) For **Description**, enter a description.
6. Choose **Generate certificate** to generate the certificate. API Gateway generates a new certificate and returns the new certificate GUID, along with the PEM-encoded public key.

You're now ready to configure an API to use the certificate.

Configure an API to use SSL certificates

These instructions assume that you already completed Generate a client certificate using the API Gateway console.

1. In the API Gateway console, create or open an API for which you want to use the client certificate. Make sure that the API has been deployed to a stage.
2. In the main navigation pane, choose **Stages**.
3. In the **Stage details** section, choose **Edit**.
4. For **Client certificate**, select a certificate.

5. Choose **Save changes**.

   If the API has been deployed previously in the API Gateway console, you'll need to redeploy it for the changes to take effect. For more information, see the section called "Redeploy a REST API to a stage".

After a certificate is selected for the API and saved, API Gateway uses the certificate for all calls to HTTP integrations in your API.

Test invoke to verify the client certificate configuration

1. Choose an API method. Choose the **Test** tab. You might need to choose the right arrow button to show the **Test** tab.
2. For **Client certificate**, select a certificate.
3. Choose **Test**.

API Gateway presents the chosen SSL certificate for the HTTP backend to authenticate the API.

Configure a backend HTTPS server to verify the client certificate

These instructions assume that you already completed Generate a client certificate using the API Gateway console and downloaded a copy of the client certificate. You can download a client certificate by calling `clientcertificate:by-id` of the API Gateway REST API or `get-client-certificate` of AWS CLI.

Before configuring a backend HTTPS server to verify the client SSL certificate of API Gateway, you must have obtained the PEM-encoded private key and a server-side certificate that is provided by a trusted certificate authority.

If the server domain name is `myserver.mydomain.com`, the server certificate's CNAME value must be `myserver.mydomain.com` or `*.mydomain.com`.

Supported certificate authorities include Let's Encrypt or one of the section called "Supported certificate authorities for HTTP and HTTP proxy integration".

As an example, suppose that the client certificate file is `apig-cert.pem` and the server private key and certificate files are `server-key.pem` and `server-cert.pem`, respectively. For a Node.js server in the backend, you can configure the server similar to the following:

```
var fs = require('fs');
var https = require('https');
var options = {
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-cert.pem'),
    ca: fs.readFileSync('apig-cert.pem'),
    requestCert: true,
    rejectUnauthorized: true
};
https.createServer(options, function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
}).listen(443);
```

For a node-express app, you can use the client-certificate-auth modules to authenticate client requests with PEM-encoded certificates.

For other HTTPS server, see the documentation for the server.

Rotate an expiring client certificate

The client certificate generated by API Gateway is valid for 365 days. You must rotate the certificate before a client certificate on an API stage expires to avoid any downtime for the API. You can check the expiration date of certificate by calling clientCertificate:by-id of the API Gateway REST API or the AWS CLI command of get-client-certificate and inspecting the returned expirationDate property.

To rotate a client certificate, do the following:

1. Generate a new client certificate by calling clientcertificate:generate of the API Gateway REST API or the AWS CLI command of generate-client-certificate. In this tutorial, we assume that the new client certificate ID is `ndiqef`.
2. Update the backend server to include the new client certificate. Don't remove the existing client certificate yet.

   Some servers might require a restart to finish the update. Consult the server documentation to see if you must restart the server during the update.
3. Update the API stage to use the new client certificate by calling stage:update of the API Gateway REST API, with the new client certificate ID (`ndiqef`):

```
PATCH /restapis/{restapi-id}/stages/stage1 HTTP/1.1
Content-Type: application/json
Host: apigateway.us-east-1.amazonaws.com
X-Amz-Date: 20170603T200400Z
Authorization: AWS4-HMAC-SHA256 Credential=...

{
  "patchOperations" : [
    {
        "op" : "replace",
        "path" : "/clientCertificateId",
        "value" : "ndiqef"
    }
  ]
}
```

or by calling the CLI command of update-stage.

4. Update the backend server to remove the old certificate.
5. Delete the old certificate from API Gateway by calling the clientcertificate:delete of the API Gateway REST API, specifying the clientCertificateId (`a1b2c3`) of the old certificate:

```
DELETE /clientcertificates/a1b2c3
```

or by calling the CLI command of delete-client-certificate:

```
aws apigateway delete-client-certificate --client-certificate-id a1b2c3
```

To rotate a client certificate in the console for a previously deployed API, do the following:

1. In the main navigation pane, choose **Client certificates**.
2. From the **Client certificates** pane, choose **Generate certificate**.
3. Open the API for which you want to use the client certificate.
4. Choose **Stages** under the selected API and then choose a stage.
5. In the **Stage details** section, choose **Edit**.
6. For **Client certificate**, select the new certificate.
7. To save the settings, choose **Save changes**.

You need to redeploy the API for the changes to take effect. For more information, see the section called "Redeploy a REST API to a stage".

API Gateway-supported certificate authorities for HTTP and HTTP proxy integrations

The following list shows the certificate authorities supported by API Gateway for HTTP, HTTP proxy, and private integrations.

```
Alias name: accvraiz1
  SHA1: 93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
  SHA256:
 9A:6E:C0:12:E1:A7:DA:9D:BE:34:19:4D:47:8A:D7:C0:DB:18:22:FB:07:1D:F1:29:81:49:6E:D1:04:38:41:1
Alias name: acraizfnmtrcm
  SHA1: EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
  SHA256:
 EB:C5:57:0C:29:01:8C:4D:67:B1:AA:12:7B:AF:12:F7:03:B4:61:1E:BC:17:B7:DA:B5:57:38:94:17:9B:93:F
Alias name: actalis
  SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
  SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: actalisauthenticationrootca
  SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
  SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: addtrustclass1ca
  SHA1: CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
  SHA256:
 8C:72:09:27:9A:C0:4E:27:5E:16:D0:7F:D3:B7:75:E8:01:54:B5:96:80:46:E3:1F:52:DD:25:76:63:24:E9:A
Alias name: addtrustexternalca
  SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
  SHA256:
 68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F
Alias name: addtrustqualifiedca
  SHA1: 4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
  SHA256:
 80:95:21:08:05:DB:4B:BC:35:5E:44:28:D8:FD:6E:C2:CD:E3:AB:5F:B9:7A:99:42:98:8E:B8:F4:DC:D0:60:1
Alias name: affirmtrustcommercial
  SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
  SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
```

```
Alias name: affirmtrustcommercialca
  SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
  SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
Alias name: affirmtrustnetworking
  SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
  SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: affirmtrustnetworkingca
  SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
  SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: affirmtrustpremium
  SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
  SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: affirmtrustpremiumca
  SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
  SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: affirmtrustpremiumecc
  SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
  SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: affirmtrustpremiumeccca
  SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
  SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: amazon-ca-g4-acm1
  SHA1: F2:0D:28:B6:29:C2:2C:5E:84:05:E6:02:4D:97:FE:8F:A0:84:93:A0
  SHA256:
 B0:11:A4:F7:29:6C:74:D8:2B:F5:62:DF:87:D7:28:C7:1F:B5:8C:F4:E6:73:F2:78:FC:DA:F3:FF:83:A6:8C:8
Alias name: amazon-ca-g4-acm2
  SHA1: A7:E6:45:32:1F:7A:B7:AD:C0:70:EA:73:5F:AB:ED:C3:DA:B4:D0:C8
  SHA256:
 D7:A8:7C:69:95:D0:E2:04:2A:32:70:A7:E2:87:FE:A7:E8:F4:C1:70:62:F7:90:C3:EB:BB:53:F2:AC:39:26:B
Alias name: amazon-ca-g4-acm3
  SHA1: 7A:DB:56:57:5F:D6:EE:67:85:0A:64:BB:1C:E9:E4:B0:9A:DB:9D:07
  SHA256:
 6B:EB:9D:20:2E:C2:00:70:BD:D2:5E:D3:C0:C8:33:2C:B4:78:07:C5:82:94:4E:7E:23:28:22:71:A4:8E:0E:C
Alias name: amazon-ca-g4-legacy
  SHA1: EA:E7:DE:F9:0A:BE:9F:0B:68:CE:B7:24:0D:80:74:03:BF:6E:B1:6E
  SHA256:
 CD:72:C4:7F:B4:AD:28:A4:67:2B:E1:86:47:D4:40:E9:3B:16:2D:95:DB:3C:2F:94:BB:81:D9:09:F7:91:24:5
```

```
Alias name: amazon-root-ca-ecc-384-1
  SHA1: F9:5E:4A:AB:9C:2D:57:61:63:3D:B2:57:B4:0F:24:9E:7B:E2:23:7D
  SHA256:
 C6:BD:E5:66:C2:72:2A:0E:96:E9:C1:2C:BF:38:92:D9:55:4D:29:03:57:30:72:40:7F:4E:70:17:3B:3C:9B:6
Alias name: amazon-root-ca-rsa-2k-1
  SHA1: 8A:9A:AC:27:FC:86:D4:50:23:AD:D5:63:F9:1E:AE:2C:AF:63:08:6C
  SHA256:
 0F:8F:33:83:FB:70:02:89:49:24:E1:AA:B0:D7:FB:5A:BF:98:DF:75:8E:0F:FE:61:86:92:BC:F0:75:35:CC:8
Alias name: amazon-root-ca-rsa-4k-1
  SHA1: EC:BD:09:61:F5:7A:B6:A8:76:BB:20:8F:14:05:ED:7E:70:ED:39:45
  SHA256:
 36:AE:AD:C2:6A:60:07:90:6B:83:A3:73:2D:D1:2B:D4:00:5E:C7:F2:76:11:99:A9:D4:DA:63:2F:59:B2:8B:C
Alias name: amazon1
  SHA1: 8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
  SHA256:
 8E:CD:E6:88:4F:3D:87:B1:12:5B:A3:1A:C3:FC:B1:3D:70:16:DE:7F:57:CC:90:4F:E1:CB:97:C6:AE:98:19:6
Alias name: amazon2
  SHA1: 5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
  SHA256:
 1B:A5:B2:AA:8C:65:40:1A:82:96:01:18:F8:0B:EC:4F:62:30:4D:83:CE:C4:71:3A:19:C3:9C:01:1E:A4:6D:B
Alias name: amazon3
  SHA1: 0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
  SHA256:
 18:CE:6C:FE:7B:F1:4E:60:B2:E3:47:B8:DF:E8:68:CB:31:D0:2E:BB:3A:DA:27:15:69:F5:03:43:B4:6D:B3:A
Alias name: amazon4
  SHA1: F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
  SHA256:
 E3:5D:28:41:9E:D0:20:25:CF:A6:90:38:CD:62:39:62:45:8D:A5:C6:95:FB:DE:A3:C2:2B:0B:FB:25:89:70:9
Alias name: amazonrootca1
  SHA1: 8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
  SHA256:
 8E:CD:E6:88:4F:3D:87:B1:12:5B:A3:1A:C3:FC:B1:3D:70:16:DE:7F:57:CC:90:4F:E1:CB:97:C6:AE:98:19:6
Alias name: amazonrootca2
  SHA1: 5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
  SHA256:
 1B:A5:B2:AA:8C:65:40:1A:82:96:01:18:F8:0B:EC:4F:62:30:4D:83:CE:C4:71:3A:19:C3:9C:01:1E:A4:6D:B
Alias name: amazonrootca3
  SHA1: 0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
  SHA256:
 18:CE:6C:FE:7B:F1:4E:60:B2:E3:47:B8:DF:E8:68:CB:31:D0:2E:BB:3A:DA:27:15:69:F5:03:43:B4:6D:B3:A
Alias name: amazonrootca4
  SHA1: F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
  SHA256:
 E3:5D:28:41:9E:D0:20:25:CF:A6:90:38:CD:62:39:62:45:8D:A5:C6:95:FB:DE:A3:C2:2B:0B:FB:25:89:70:9
```

```
Alias name: amzninternalinfoseccag3
  SHA1: B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
  SHA256:
 81:03:0B:C7:E2:54:DA:7B:F8:B7:45:DB:DD:41:15:89:B5:A3:81:86:FB:4B:29:77:1F:84:0A:18:D9:67:6D:6
Alias name: amzninternalrootca
  SHA1: A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
  SHA256:
 0E:DE:63:C1:DC:7A:8E:11:F1:AB:BC:05:4F:59:EE:49:9D:62:9A:2F:DE:9C:A7:16:32:A2:64:29:3E:8B:66:A
Alias name: aolrootca1
  SHA1: 39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
  SHA256:
 77:40:73:12:C6:3A:15:3D:5B:C0:0B:4E:51:75:9C:DF:DA:C2:37:DC:2A:33:B6:79:46:E9:8E:9B:FA:68:0A:E
Alias name: aolrootca2
  SHA1: 85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
  SHA256:
 7D:3B:46:5A:60:14:E5:26:C0:AF:FC:EE:21:27:D2:31:17:27:AD:81:1C:26:84:2D:00:6A:F3:73:06:CC:80:B
Alias name: atostrustedroot2011
  SHA1: 2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
  SHA256:
 F3:56:BE:A2:44:B7:A9:1E:B3:5D:53:CA:9A:D7:86:4A:CE:01:8E:2D:35:D5:F8:F9:6D:DF:68:A6:F4:1A:A4:7
Alias name: autoridaddecertificacionfirmaprofesionalcifa62634068
  SHA1: AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
  SHA256:
 04:04:80:28:BF:1F:28:64:D4:8F:9A:D4:D8:32:94:36:6A:82:88:56:55:3F:3B:14:30:3F:90:14:7F:5D:40:E
Alias name: baltimorecodesigningca
  SHA1: 30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
  SHA256:
 A9:15:45:DB:D2:E1:9C:4C:CD:F9:09:AA:71:90:0D:18:C7:35:1C:89:B3:15:F0:F1:3D:05:C1:3A:8F:FB:46:8
Alias name: baltimorecybertrustca
  SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
  SHA256:
 16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: baltimorecybertrustroot
  SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
  SHA256:
 16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: buypassclass2ca
  SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
  SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
Alias name: buypassclass2rootca
  SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
  SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
```

```
Alias name: buypassclass3ca
   SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
   SHA256:
 ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: buypassclass3rootca
   SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
   SHA256:
 ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: cadisigrootr2
   SHA1: B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
   SHA256:
 E2:3D:4A:03:6D:7B:70:E9:F5:95:B1:42:20:79:D2:B9:1E:DF:BB:1F:B6:51:A0:63:3E:AA:8A:9D:C5:F8:07:0
Alias name: camerfirmachambersca
   SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
   SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: camerfirmachamberscommerceca
   SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
   SHA256:
 0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C
Alias name: camerfirmachambersignca
   SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
   SHA256:
 13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
Alias name: certigna
   SHA1: B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
   SHA256:
 E3:B6:A2:DB:2E:D7:CE:48:84:2F:7A:C5:32:41:C7:B7:1D:54:14:4B:FB:40:C1:1F:3F:1D:0B:42:F5:EE:A1:2
Alias name: certignarootca
   SHA1: 2D:0D:52:14:FF:9E:AD:99:24:01:74:20:47:6E:6C:85:27:27:F5:43
   SHA256:
 D4:8D:3D:23:EE:DB:50:A4:59:E5:51:97:60:1C:27:77:4B:9D:7B:18:C9:4D:5A:05:95:11:A1:02:50:B9:31:6
Alias name: certplusclass2primaryca
   SHA1: 74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
   SHA256:
 0F:99:3C:8A:EF:97:BA:AF:56:87:14:0E:D5:9A:D1:82:1B:B4:AF:AC:F0:AA:9A:58:B5:D5:7A:33:8A:3A:FB:C
Alias name: certplusclass3pprimaryca
   SHA1: 21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
   SHA256:
 CC:C8:94:89:37:1B:AD:11:1C:90:61:9B:EA:24:0A:2E:6D:AD:D9:9F:9F:6E:1D:4D:41:E5:8E:D6:DE:3D:02:8
Alias name: certsignrootca
   SHA1: FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
   SHA256:
 EA:A9:62:C4:FA:4A:6B:AF:EB:E4:15:19:6D:35:1C:CD:88:8D:4F:53:F3:FA:8A:E6:D7:C4:66:A9:4E:60:42:B
```

```
Alias name: certsignrootcag2
  SHA1: 26:F9:93:B4:ED:3D:28:27:B0:B9:4B:A7:E9:15:1D:A3:8D:92:E5:32
  SHA256:
 65:7C:FE:2F:A7:3F:AA:38:46:25:71:F3:32:A2:36:3A:46:FC:E7:02:09:51:71:07:02:CD:FB:B6:EE:DA:33:0
Alias name: certum2
  SHA1: D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
  SHA256:
 B6:76:F2:ED:DA:E8:77:5C:D3:6C:B0:F6:3C:D1:D4:60:39:61:F4:9E:62:65:BA:01:3A:2F:03:07:B6:D0:B8:0
Alias name: certumca
  SHA1: 62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
  SHA256:
 D8:E0:FE:BC:1D:B2:E3:8D:00:94:0F:37:D2:7D:41:34:4D:99:3E:73:4B:99:D5:65:6D:97:78:D4:D8:14:36:2
Alias name: certumtrustednetworkca
  SHA1: 07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
  SHA256:
 5C:58:46:8D:55:F5:8E:49:7E:74:39:82:D2:B5:00:10:B6:D1:65:37:4A:CF:83:A7:D4:A3:2D:B7:68:C4:40:8
Alias name: certumtrustednetworkca2
  SHA1: D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
  SHA256:
 B6:76:F2:ED:DA:E8:77:5C:D3:6C:B0:F6:3C:D1:D4:60:39:61:F4:9E:62:65:BA:01:3A:2F:03:07:B6:D0:B8:0
Alias name: cfcaevroot
  SHA1: E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
  SHA256:
 5C:C3:D7:8E:4E:1D:5E:45:54:7A:04:E6:87:3E:64:F9:0C:F9:53:6D:1C:CC:2E:F8:00:F3:55:C4:C5:FD:70:F
Alias name: chambersofcommerceroot2008
  SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
  SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: chunghwaepkirootca
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
  SHA256:
 C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
Alias name: cia-crt-g3-01-ca
  SHA1: 2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
  SHA256:
 20:48:AD:4C:EC:90:7F:FA:4A:15:D4:CE:45:E3:C8:E4:2C:EA:78:33:DC:C7:D3:40:48:FC:60:47:27:42:99:E
Alias name: cia-crt-g3-02-ca
  SHA1: 96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
  SHA256:
 93:F1:72:FB:BA:43:31:5C:06:EE:0F:9F:04:89:B8:F6:88:BC:75:15:3C:BE:B4:80:AC:A7:14:3A:F6:FC:4A:C
Alias name: comodo-ca
  SHA1: AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
  SHA256:
 52:F0:E1:C4:E5:8E:C6:29:29:1B:60:31:7F:07:46:71:B8:5D:7E:A8:0D:5B:07:27:34:63:53:4B:32:B4:02:3
```

```
Alias name: comodoaaaca
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
Alias name: comodoaaaservicesroot
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
Alias name: comodocertificationauthority
  SHA1: 66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
  SHA256:
 0C:2C:D6:3D:F7:80:6F:A3:99:ED:E8:09:11:6B:57:5B:F8:79:89:F0:65:18:F9:80:8C:86:05:03:17:8B:AF:6
Alias name: comodoecccertificationauthority
  SHA1: 9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
  SHA256:
 17:93:92:7A:06:14:54:97:89:AD:CE:2F:8F:34:F7:F0:B6:6D:0F:3A:E3:A3:B8:4D:21:EC:15:DB:BA:4F:AD:C
Alias name: comodorsacertificationauthority
  SHA1: AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
  SHA256:
 52:F0:E1:C4:E5:8E:C6:29:29:1B:60:31:7F:07:46:71:B8:5D:7E:A8:0D:5B:07:27:34:63:53:4B:32:B4:02:3
Alias name: cybertrustglobalroot
  SHA1: 5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
  SHA256:
 96:0A:DF:00:63:E9:63:56:75:0C:29:65:DD:0A:08:67:DA:0B:9C:BD:6E:77:71:4A:EA:FB:23:49:AB:39:3D:A
Alias name: deprecateditsecca
  SHA1: 12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
  SHA256:
 9A:59:DA:86:24:1A:FD:BA:A3:39:FA:9C:FD:21:6A:0B:06:69:4D:E3:7E:37:52:6B:BE:63:C8:BC:83:74:2E:C
Alias name: deutschetelekomrootca2
  SHA1: 85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
  SHA256:
 B6:19:1A:50:D0:C3:97:7F:7D:A9:9B:CD:AA:C8:6A:22:7D:AE:B9:67:9E:C7:0B:A3:B0:C9:D9:22:71:C1:70:D
Alias name: digicertassuredidrootca
  SHA1: 05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
  SHA256:
 3E:90:99:B5:01:5E:8F:48:6C:00:BC:EA:9D:11:1E:E7:21:FA:BA:35:5A:89:BC:F1:DF:69:56:1E:3D:C6:32:5
Alias name: digicertassuredidrootg2
  SHA1: A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
  SHA256:
 7D:05:EB:B6:82:33:9F:8C:94:51:EE:09:4E:EB:FE:FA:79:53:A1:14:ED:B2:F4:49:49:45:2F:AB:7D:2F:C1:8
Alias name: digicertassuredidrootg3
  SHA1: F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
  SHA256:
 7E:37:CB:8B:4C:47:09:0C:AB:36:55:1B:A6:F4:5D:B8:40:68:0F:BA:16:6A:95:2D:B1:00:71:7F:43:05:3F:C
```

```
Alias name: digicertglobalrootca
  SHA1: A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
  SHA256:
 43:48:A0:E9:44:4C:78:CB:26:5E:05:8D:5E:89:44:B4:D8:4F:96:62:BD:26:DB:25:7F:89:34:A4:43:C7:01:6
Alias name: digicertglobalrootg2
  SHA1: DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
  SHA256:
 CB:3C:CB:B7:60:31:E5:E0:13:8F:8D:D3:9A:23:F9:DE:47:FF:C3:5E:43:C1:14:4C:EA:27:D4:6A:5A:B1:CB:5
Alias name: digicertglobalrootg3
  SHA1: 7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
  SHA256:
 31:AD:66:48:F8:10:41:38:C7:38:F3:9E:A4:32:01:33:39:3E:3A:18:CC:02:29:6E:F9:7C:2A:C9:EF:67:31:D
Alias name: digicerthighassuranceevrootca
  SHA1: 5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
  SHA256:
 74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:C
Alias name: digicerttrustedrootg4
  SHA1: DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
  SHA256:
 55:2F:7B:DC:F1:A7:AF:9E:6C:E6:72:01:7F:4F:12:AB:F7:72:40:C7:8E:76:1A:C2:03:D1:D9:D2:0A:C8:99:8
Alias name: dstrootcax3
  SHA1: DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
  SHA256:
 06:87:26:03:31:A7:24:03:D9:09:F1:05:E6:9B:CF:0D:32:E1:BD:24:93:FF:C6:D9:20:6D:11:BC:D6:77:07:3
Alias name: dtrustrootclass3ca22009
  SHA1: 58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
  SHA256:
 49:E7:A4:42:AC:F0:EA:62:87:05:00:54:B5:25:64:B6:50:E4:F4:9E:42:E3:48:D6:AA:38:E0:39:E9:57:B1:C
Alias name: dtrustrootclass3ca2ev2009
  SHA1: 96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
  SHA256:
 EE:C5:49:6B:98:8C:E9:86:25:B9:34:09:2E:EC:29:08:BE:D0:B0:F3:16:C2:D4:73:0C:84:EA:F1:F3:D3:48:8
Alias name: ecacc
  SHA1: 28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
  SHA256:
 88:49:7F:01:60:2F:31:54:24:6A:E2:8C:4D:5A:EF:10:F1:D8:7E:BB:76:62:6F:4A:E0:B7:F9:5B:A7:96:87:9
Alias name: emsigneccrootcac3
  SHA1: B6:AF:43:C2:9B:81:53:7D:F6:EF:6B:C3:1F:1F:60:15:0C:EE:48:66
  SHA256:
 BC:4D:80:9B:15:18:9D:78:DB:3E:1D:8C:F4:F9:72:6A:79:5D:A1:64:3C:A5:F1:35:8E:1D:DB:0E:DC:0D:7E:B
Alias name: emsigneccrootcag3
  SHA1: 30:43:FA:4F:F2:57:DC:A0:C3:80:EE:2E:58:EA:78:B2:3F:E6:BB:C1
  SHA256:
 86:A1:EC:BA:08:9C:4A:8D:3B:BE:27:34:C6:12:BA:34:1D:81:3E:04:3C:F9:E8:A8:62:CD:5C:57:A3:6B:BE:6
```

```
Alias name: emsignrootcac1
  SHA1: E7:2E:F1:DF:FC:B2:09:28:CF:5D:D4:D5:67:37:B1:51:CB:86:4F:01
  SHA256:
 12:56:09:AA:30:1D:A0:A2:49:B9:7A:82:39:CB:6A:34:21:6F:44:DC:AC:9F:39:54:B1:42:92:F2:E8:C8:60:8
Alias name: emsignrootcag1
  SHA1: 8A:C7:AD:8F:73:AC:4E:C1:B5:75:4D:A5:40:F4:FC:CF:7C:B5:8E:8C
  SHA256:
 40:F6:AF:03:46:A9:9A:A1:CD:1D:55:5A:4E:9C:CE:62:C7:F9:63:46:03:EE:40:66:15:83:3D:C8:C8:D0:03:6
Alias name: entrust2048ca
  SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
  SHA256:
 6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: entrustevca
  SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
  SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: entrustnetpremium2048secureserverca
  SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
  SHA256:
 6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: entrustrootcag2
  SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
  SHA256:
 43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:3
Alias name: entrustrootcertificationauthority
  SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
  SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: entrustrootcertificationauthorityec1
  SHA1: 20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
  SHA256:
 02:ED:0E:B2:8C:14:DA:45:16:5C:56:67:91:70:0D:64:51:D7:FB:56:F0:B2:AB:1D:3B:8E:B0:70:E5:6E:DF:F
Alias name: entrustrootcertificationauthorityg2
  SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
  SHA256:
 43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:3
Alias name: entrustrootcertificationauthorityg4
  SHA1: 14:88:4E:86:26:37:B0:26:AF:59:62:5C:40:77:EC:35:29:BA:96:01
  SHA256:
 DB:35:17:D1:F6:73:2A:2D:5A:B9:7C:53:3E:C7:07:79:EE:32:70:A6:2F:B4:AC:42:38:37:24:60:E6:F0:1E:8
Alias name: epkirootcertificationauthority
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
  SHA256:
 C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
```

```
Alias name: equifaxsecureebusinessca1
  SHA1: AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
  SHA256:
 2E:3A:2B:B5:11:25:05:83:6C:A8:96:8B:E2:CB:37:27:CE:9B:56:84:5C:6E:E9:8E:91:85:10:4A:FB:9A:F5:9
Alias name: equifaxsecureglobalebusinessca1
  SHA1: 3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
  SHA256:
 86:AB:5A:65:71:D3:32:9A:BC:D2:E4:E6:37:66:8B:A8:9C:73:1E:C2:93:B6:CB:A6:0F:71:63:40:A0:91:CE:A
Alias name: eszignorootca2017
  SHA1: 89:D4:83:03:4F:9E:9A:48:80:5F:72:37:D4:A9:A6:EF:CB:7C:1F:D1
  SHA256:
 BE:B0:0B:30:83:9B:9B:C3:2C:32:E4:44:79:05:95:06:41:F2:64:21:B1:5E:D0:89:19:8B:51:8A:E2:EA:1B:9
Alias name: etugracertificationauthority
  SHA1: 51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
  SHA256:
 B0:BF:D5:2B:B0:D7:D9:BD:92:BF:5D:4D:C1:3D:A2:55:C0:2C:54:2F:37:83:65:EA:89:39:11:F5:5E:55:F2:3
Alias name: gd-class2-root.pem
  SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
  SHA256:
 C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
Alias name: gd_bundle-g2.pem
  SHA1: 27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
  SHA256:
 97:3A:41:27:6F:FD:01:E0:27:A2:AA:D4:9E:34:C3:78:46:D3:E9:76:FF:6A:62:0B:67:12:E3:38:32:04:1A:A
Alias name: gdcatrustauthr5root
  SHA1: 0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
  SHA256:
 BF:FF:8F:D0:44:33:48:7D:6A:8A:A6:0C:1A:29:76:7A:9F:C2:BB:B0:5E:42:0F:71:3A:13:B9:92:89:1D:38:9
Alias name: gdroot-g2.pem
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: geotrustglobalca
  SHA1: DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
  SHA256:
 FF:85:6A:2D:25:1D:CD:88:D3:66:56:F4:50:12:67:98:CF:AB:AA:DE:40:79:9C:72:2D:E4:D2:B5:DB:36:A7:3
Alias name: geotrustprimaryca
  SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
  SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: geotrustprimarycag2
  SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
  SHA256:
 5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
```

```
Alias name: geotrustprimarycag3
  SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
  SHA256:
 B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
Alias name: geotrustprimarycertificationauthority
  SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
  SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: geotrustprimarycertificationauthorityg2
  SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
  SHA256:
 5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
Alias name: geotrustprimarycertificationauthorityg3
  SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
  SHA256:
 B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
Alias name: geotrustuniversalca
  SHA1: E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
  SHA256:
 A0:45:9B:9F:63:B2:25:59:F5:FA:5D:4C:6D:B3:F9:F7:2F:F1:93:42:03:35:78:F0:73:BF:1D:1B:46:CB:B9:1
Alias name: geotrustuniversalca2
  SHA1: 37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
  SHA256:
 A0:23:4F:3B:C8:52:7C:A5:62:8E:EC:81:AD:5D:69:89:5D:A5:68:0D:C9:1D:1C:B8:47:7F:33:F8:78:B9:5B:0
Alias name: globalchambersignroot2008
  SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
  SHA256:
 13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
Alias name: globalsignca
  SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
  SHA256:
 EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
Alias name: globalsigneccrootcar4
  SHA1: 69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
  SHA256:
 BE:C9:49:11:C2:95:56:76:DB:6C:0A:55:09:86:D7:6E:3B:A0:05:66:7C:44:2C:97:62:B4:FB:B7:73:DE:22:8
Alias name: globalsigneccrootcar5
  SHA1: 1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
  SHA256:
 17:9F:BC:14:8A:3D:D0:0F:D2:4E:A1:34:58:CC:43:BF:A7:F5:9C:81:82:D7:83:A5:13:F6:EB:EC:10:0C:89:2
Alias name: globalsignr2ca
  SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
  SHA256:
 CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
```

```
Alias name: globalsignr3ca
  SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
  SHA256:
 CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
Alias name: globalsignrootca
  SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
  SHA256:
 EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
Alias name: globalsignrootcar2
  SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
  SHA256:
 CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
Alias name: globalsignrootcar3
  SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
  SHA256:
 CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
Alias name: globalsignrootcar6
  SHA1: 80:94:64:0E:B5:A7:A1:CA:11:9C:1F:DD:D5:9F:81:02:63:A7:FB:D1
  SHA256:
 2C:AB:EA:FE:37:D0:6C:A2:2A:BA:73:91:C0:03:3D:25:98:29:52:C4:53:64:73:49:76:3A:3A:B5:AD:6C:CF:6
Alias name: godaddyclass2ca
  SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
  SHA256:
 C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
Alias name: godaddyrootcertificateauthorityg2
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: godaddyrootg2ca
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: gtsrootr1
  SHA1: E1:C9:50:E6:EF:22:F8:4C:56:45:72:8B:92:20:60:D7:D5:A7:A3:E8
  SHA256:
 2A:57:54:71:E3:13:40:BC:21:58:1C:BD:2C:F1:3E:15:84:63:20:3E:CE:94:BC:F9:D3:CC:19:6B:F0:9A:54:7
Alias name: gtsrootr2
  SHA1: D2:73:96:2A:2A:5E:39:9F:73:3F:E1:C7:1E:64:3F:03:38:34:FC:4D
  SHA256:
 C4:5D:7B:B0:8E:6D:67:E6:2E:42:35:11:0B:56:4E:5F:78:FD:92:EF:05:8C:84:0A:EA:4E:64:55:D7:58:5C:6
Alias name: gtsrootr3
  SHA1: 30:D4:24:6F:07:FF:DB:91:89:8A:0B:E9:49:66:11:EB:8C:5E:46:E5
  SHA256:
 15:D5:B8:77:46:19:EA:7D:54:CE:1C:A6:D0:B0:C4:03:E0:37:A9:17:F1:31:E8:A0:4E:1E:6B:7A:71:BA:BC:E
```

```
Alias name: gtsrootr4
  SHA1: 2A:1D:60:27:D9:4A:B1:0A:1C:4D:91:5C:CD:33:A0:CB:3E:2D:54:CB
  SHA256:
 71:CC:A5:39:1F:9E:79:4B:04:80:25:30:B3:63:E1:21:DA:8A:30:43:BB:26:66:2F:EA:4D:CA:7F:C9:51:A4:B
Alias name: hellenicacademicandresearchinstitutionseccrootca2015
  SHA1: 9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
  SHA256:
 44:B5:45:AA:8A:25:E6:5A:73:CA:15:DC:27:FC:36:D2:4C:1C:B9:95:3A:06:65:39:B1:15:82:DC:48:7B:48:3
Alias name: hellenicacademicandresearchinstitutionsrootca2011
  SHA1: FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
  SHA256:
 BC:10:4F:15:A4:8B:E7:09:DC:A5:42:A7:E1:D4:B9:DF:6F:05:45:27:E8:02:EA:A9:2D:59:54:44:25:8A:FE:7
Alias name: hellenicacademicandresearchinstitutionsrootca2015
  SHA1: 01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
  SHA256:
 A0:40:92:9A:02:CE:53:B4:AC:F4:F2:FF:C6:98:1C:E4:49:6F:75:5E:6D:45:FE:0B:2A:69:2B:CD:52:52:3F:3
Alias name: hongkongpostrootca1
  SHA1: D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
  SHA256:
 F9:E6:7D:33:6C:51:00:2A:C0:54:C6:32:02:2D:66:DD:A2:E7:E3:FF:F1:0A:D0:61:ED:31:D8:BB:B4:10:CF:B
Alias name: hongkongpostrootca3
  SHA1: 58:A2:D0:EC:20:52:81:5B:C1:F3:F8:64:02:24:4E:C2:8E:02:4B:02
  SHA256:
 5A:2F:C0:3F:0C:83:B0:90:BB:FA:40:60:4B:09:88:44:6C:76:36:18:3D:F9:84:6E:17:10:1A:44:7F:B8:EF:D
Alias name: identrustcommercialrootca1
  SHA1: DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
  SHA256:
 5D:56:49:9B:E4:D2:E0:8B:CF:CA:D0:8A:3E:38:72:3D:50:50:3B:DE:70:69:48:E4:2F:55:60:30:19:E5:28:A
Alias name: identrustpublicsectorrootca1
  SHA1: BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
  SHA256:
 30:D0:89:5A:9A:44:8A:26:20:91:63:55:22:D1:F5:20:10:B5:86:7A:CA:E1:2C:78:EF:95:8F:D4:F4:38:9F:2
Alias name: isrgrootx1
  SHA1: CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
  SHA256:
 96:BC:EC:06:26:49:76:F3:74:60:77:9A:CF:28:C5:A7:CF:E8:A3:C0:AA:E1:1A:8F:FC:EE:05:C0:BD:DF:08:C
Alias name: izenpecom
  SHA1: 2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
  SHA256:
 25:30:CC:8E:98:32:15:02:BA:D9:6F:9B:1F:BA:1B:09:9E:2D:29:9E:0F:45:48:BB:91:4F:36:3B:C0:D4:53:1
Alias name: keynectisrootca
  SHA1: 9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
  SHA256:
 42:10:F1:99:49:9A:9A:C3:3C:8D:E0:2B:A6:DB:AA:14:40:8B:DD:8A:6E:32:46:89:C1:92:2D:06:97:15:A3:3
```

```
Alias name: microseceszignorootca2009
  SHA1: 89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
  SHA256:
 3C:5F:81:FE:A5:FA:B8:2C:64:BF:A2:EA:EC:AF:CD:E8:E0:77:FC:86:20:A7:CA:E5:37:16:3D:F3:6E:DB:F3:7
Alias name: mozillacert0.pem
  SHA1: 97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
  SHA256:
 A5:31:25:18:8D:21:10:AA:96:4B:02:C7:B7:C6:DA:32:03:17:08:94:E5:FB:71:FF:FB:66:67:D5:E6:81:0A:3
Alias name: mozillacert1.pem
  SHA1: 23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
  SHA256:
 B4:41:0B:73:E2:E6:EA:CA:47:FB:C4:2F:8F:A4:01:8A:F4:38:1D:C5:4C:FA:A8:44:50:46:1E:ED:09:45:4D:E
Alias name: mozillacert10.pem
  SHA1: 5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
  SHA256:
 21:DB:20:12:36:60:BB:2E:D4:18:20:5D:A1:1E:E7:A8:5A:65:E2:BC:6E:55:B5:AF:7E:78:99:C8:A2:66:D9:2
Alias name: mozillacert100.pem
  SHA1: 58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
  SHA256:
 49:E7:A4:42:AC:F0:EA:62:87:05:00:54:B5:25:64:B6:50:E4:F4:9E:42:E3:48:D6:AA:38:E0:39:E9:57:B1:C
Alias name: mozillacert101.pem
  SHA1: 99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
  SHA256:
 62:F2:40:27:8C:56:4C:4D:D8:BF:7D:9D:4F:6F:36:6E:A8:94:D2:2F:5F:34:D9:89:A9:83:AC:EC:2F:FF:ED:5
Alias name: mozillacert102.pem
  SHA1: 96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
  SHA256:
 EE:C5:49:6B:98:8C:E9:86:25:B9:34:09:2E:EC:29:08:BE:D0:B0:F3:16:C2:D4:73:0C:84:EA:F1:F3:D3:48:8
Alias name: mozillacert103.pem
  SHA1: 70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
  SHA256:
 3C:FC:3C:14:D1:F6:84:FF:17:E3:8C:43:CA:44:0C:00:B9:67:EC:93:3E:8B:FE:06:4C:A1:D7:2C:90:F2:AD:B
Alias name: mozillacert104.pem
  SHA1: 4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
  SHA256:
 1C:01:C6:F4:DB:B2:FE:FC:22:55:8B:2B:CA:32:56:3F:49:84:4A:CF:C3:2B:7B:E4:B0:FF:59:9F:9E:8C:7A:F
Alias name: mozillacert105.pem
  SHA1: 77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
  SHA256:
 F0:9B:12:2C:71:14:F4:A0:9B:D4:EA:4F:4A:99:D5:58:B4:6E:4C:25:CD:81:14:0D:29:C0:56:13:91:4C:38:4
Alias name: mozillacert106.pem
  SHA1: E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
  SHA256:
 D9:5F:EA:3C:A4:EE:DC:E7:4C:D7:6E:75:FC:6D:1F:F6:2C:44:1F:0F:A8:BC:77:F0:34:B1:9E:5D:B2:58:01:5
```

```
Alias name: mozillacert107.pem
   SHA1: 8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
   SHA256:
 F9:6F:23:F4:C3:E7:9C:07:7A:46:98:8D:5A:F5:90:06:76:A0:F0:39:CB:64:5D:D1:75:49:B2:16:C8:24:40:C
Alias name: mozillacert108.pem
   SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
   SHA256:
 EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
Alias name: mozillacert109.pem
   SHA1: B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
   SHA256:
 E2:3D:4A:03:6D:7B:70:E9:F5:95:B1:42:20:79:D2:B9:1E:DF:BB:1F:B6:51:A0:63:3E:AA:8A:9D:C5:F8:07:0
Alias name: mozillacert11.pem
   SHA1: 05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
   SHA256:
 3E:90:99:B5:01:5E:8F:48:6C:00:BC:EA:9D:11:1E:E7:21:FA:BA:35:5A:89:BC:F1:DF:69:56:1E:3D:C6:32:5
Alias name: mozillacert110.pem
   SHA1: 93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
   SHA256:
 9A:6E:C0:12:E1:A7:DA:9D:BE:34:19:4D:47:8A:D7:C0:DB:18:22:FB:07:1D:F1:29:81:49:6E:D1:04:38:41:1
Alias name: mozillacert111.pem
   SHA1: 9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
   SHA256:
 59:76:90:07:F7:68:5D:0F:CD:50:87:2F:9F:95:D5:75:5A:5B:2B:45:7D:81:F3:69:2B:61:0A:98:67:2F:0E:1
Alias name: mozillacert112.pem
   SHA1: 43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
   SHA256:
 DD:69:36:FE:21:F8:F0:77:C1:23:A1:A5:21:C1:22:24:F7:22:55:B7:3E:03:A7:26:06:93:E8:A2:4B:0F:A3:8
Alias name: mozillacert113.pem
   SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
   SHA256:
 6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: mozillacert114.pem
   SHA1: 51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
   SHA256:
 B0:BF:D5:2B:B0:D7:D9:BD:92:BF:5D:4D:C1:3D:A2:55:C0:2C:54:2F:37:83:65:EA:89:39:11:F5:5E:55:F2:3
Alias name: mozillacert115.pem
   SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
   SHA256:
 91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
Alias name: mozillacert116.pem
   SHA1: 2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
   SHA256:
 F3:56:BE:A2:44:B7:A9:1E:B3:5D:53:CA:9A:D7:86:4A:CE:01:8E:2D:35:D5:F8:F9:6D:DF:68:A6:F4:1A:A4:7
```

```
Alias name: mozillacert117.pem
   SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
   SHA256:
 16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: mozillacert118.pem
   SHA1: 7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
   SHA256:
 5F:0B:62:EA:B5:E3:53:EA:65:21:65:16:58:FB:B6:53:59:F4:43:28:0A:4A:FB:D1:04:D7:7D:10:F9:F0:4C:0
Alias name: mozillacert119.pem
   SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
   SHA256:
 CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
Alias name: mozillacert12.pem
   SHA1: A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
   SHA256:
 43:48:A0:E9:44:4C:78:CB:26:5E:05:8D:5E:89:44:B4:D8:4F:96:62:BD:26:DB:25:7F:89:34:A4:43:C7:01:6
Alias name: mozillacert120.pem
   SHA1: DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
   SHA256:
 CF:56:FF:46:A4:A1:86:10:9D:D9:65:84:B5:EE:B5:8A:51:0C:42:75:B0:E5:F9:4F:40:BB:AE:86:5E:19:F6:7
Alias name: mozillacert121.pem
   SHA1: CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
   SHA256:
 8C:72:09:27:9A:C0:4E:27:5E:16:D0:7F:D3:B7:75:E8:01:54:B5:96:80:46:E3:1F:52:DD:25:76:63:24:E9:A
Alias name: mozillacert122.pem
   SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
   SHA256:
 68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F
Alias name: mozillacert123.pem
   SHA1: 2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
   SHA256:
 07:91:CA:07:49:B2:07:82:AA:D3:C7:D7:BD:0C:DF:C9:48:58:35:84:3E:B2:D7:99:60:09:CE:43:AB:6C:69:2
Alias name: mozillacert124.pem
   SHA1: 4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
   SHA256:
 80:95:21:08:05:DB:4B:BC:35:5E:44:28:D8:FD:6E:C2:CD:E3:AB:5F:B9:7A:99:42:98:8E:B8:F4:DC:D0:60:1
Alias name: mozillacert125.pem
   SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
   SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: mozillacert126.pem
   SHA1: 25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
   SHA256:
 AF:8B:67:62:A1:E5:28:22:81:61:A9:5D:5C:55:9E:E2:66:27:8F:75:D7:9E:83:01:89:A5:03:50:6A:BD:6B:4
```

```
Alias name: mozillacert127.pem
  SHA1: DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
  SHA256:
 FF:85:6A:2D:25:1D:CD:88:D3:66:56:F4:50:12:67:98:CF:AB:AA:DE:40:79:9C:72:2D:E4:D2:B5:DB:36:A7:3
Alias name: mozillacert128.pem
  SHA1: A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
  SHA256:
 CA:2D:82:A0:86:77:07:2F:8A:B6:76:4F:F0:35:67:6C:FE:3E:5E:32:5E:01:21:72:DF:3F:92:09:6D:B7:9B:8
Alias name: mozillacert129.pem
  SHA1: E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
  SHA256:
 A0:45:9B:9F:63:B2:25:59:F5:FA:5D:4C:6D:B3:F9:F7:2F:F1:93:42:03:35:78:F0:73:BF:1D:1B:46:CB:B9:1
Alias name: mozillacert13.pem
  SHA1: 06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
  SHA256:
 6C:61:DA:C3:A2:DE:F0:31:50:6B:E0:36:D2:A6:FE:40:19:94:FB:D1:3D:F9:C8:D4:66:59:92:74:C4:46:EC:9
Alias name: mozillacert130.pem
  SHA1: E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
  SHA256:
 F4:C1:49:55:1A:30:13:A3:5B:C7:BF:FE:17:A7:F3:44:9B:C1:AB:5B:5A:0A:E7:4B:06:C2:3B:90:00:4C:01:0
Alias name: mozillacert131.pem
  SHA1: 37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
  SHA256:
 A0:23:4F:3B:C8:52:7C:A5:62:8E:EC:81:AD:5D:69:89:5D:A5:68:0D:C9:1D:1C:B8:47:7F:33:F8:78:B9:5B:0
Alias name: mozillacert132.pem
  SHA1: 39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
  SHA256:
 77:40:73:12:C6:3A:15:3D:5B:C0:0B:4E:51:75:9C:DF:DA:C2:37:DC:2A:33:B6:79:46:E9:8E:9B:FA:68:0A:E
Alias name: mozillacert133.pem
  SHA1: 85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
  SHA256:
 7D:3B:46:5A:60:14:E5:26:C0:AF:FC:EE:21:27:D2:31:17:27:AD:81:1C:26:84:2D:00:6A:F3:73:06:CC:80:B
Alias name: mozillacert134.pem
  SHA1: 70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
  SHA256:
 69:FA:C9:BD:55:FB:0A:C7:8D:53:BB:EE:5C:F1:D5:97:98:9F:D0:AA:AB:20:A2:51:51:BD:F1:73:3E:E7:D1:2
Alias name: mozillacert135.pem
  SHA1: 62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
  SHA256:
 D8:E0:FE:BC:1D:B2:E3:8D:00:94:0F:37:D2:7D:41:34:4D:99:3E:73:4B:99:D5:65:6D:97:78:D4:D8:14:36:2
Alias name: mozillacert136.pem
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
```

```
Alias name: mozillacert137.pem
  SHA1: 4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
  SHA256:
 BD:81:CE:3B:4F:65:91:D1:1A:67:B5:FC:7A:47:FD:EF:25:52:1B:F9:AA:4E:18:B9:E3:DF:2E:34:A7:80:3B:E
Alias name: mozillacert138.pem
  SHA1: E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
  SHA256:
 3F:06:E5:56:81:D4:96:F5:BE:16:9E:B5:38:9F:9F:2B:8F:F6:1E:17:08:DF:68:81:72:48:49:CD:5D:27:CB:6
Alias name: mozillacert139.pem
  SHA1: DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
  SHA256:
 A4:5E:DE:3B:BB:F0:9C:8A:E1:5C:72:EF:C0:72:68:D6:93:A2:1C:99:6F:D5:1E:67:CA:07:94:60:FD:6D:88:7
Alias name: mozillacert14.pem
  SHA1: 5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
  SHA256:
 74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:C
Alias name: mozillacert140.pem
  SHA1: CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
  SHA256:
 85:A0:DD:7D:D7:20:AD:B7:FF:05:F8:3D:54:2B:20:9D:C7:FF:45:28:F7:D6:77:B1:83:89:FE:A5:E5:C4:9E:8
Alias name: mozillacert141.pem
  SHA1: 31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
  SHA256:
 58:D0:17:27:9C:D4:DC:63:AB:DD:B1:96:A6:C9:90:6C:30:C4:E0:87:83:EA:E8:C1:60:99:54:D6:93:55:59:6
Alias name: mozillacert142.pem
  SHA1: 1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
  SHA256:
 18:F1:FC:7F:20:5D:F8:AD:DD:EB:7F:E0:07:DD:57:E3:AF:37:5A:9C:4D:8D:73:54:6B:F4:F1:FE:D1:E1:8D:3
Alias name: mozillacert143.pem
  SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
  SHA256:
 E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: mozillacert144.pem
  SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
  SHA256:
 79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
Alias name: mozillacert145.pem
  SHA1: 10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
  SHA256:
 D4:1D:82:9E:8C:16:59:82:2A:F9:3F:CE:62:BF:FC:DE:26:4F:C8:4E:8B:95:0C:5F:F2:75:D0:52:35:46:95:A
Alias name: mozillacert146.pem
  SHA1: 21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
  SHA256:
 48:98:C6:88:8C:0C:FF:B0:D3:E3:1A:CA:8A:37:D4:E3:51:5F:F7:46:D0:26:35:D8:66:46:CF:A0:A3:18:5A:E
```

```
Alias name: mozillacert147.pem
   SHA1: 58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
   SHA256:
 85:FB:2F:91:DD:12:27:5A:01:45:B6:36:53:4F:84:02:4A:D6:8B:69:B8:EE:88:68:4F:F7:11:37:58:05:B3:4
Alias name: mozillacert148.pem
   SHA1: 04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
   SHA256:
 6E:A5:47:41:D0:04:66:7E:ED:1B:48:16:63:4A:A3:A7:9E:6E:4B:96:95:0F:82:79:DA:FC:8D:9B:D8:81:21:3
Alias name: mozillacert149.pem
   SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
   SHA256:
 0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C
Alias name: mozillacert15.pem
   SHA1: 74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
   SHA256:
 0F:99:3C:8A:EF:97:BA:AF:56:87:14:0E:D5:9A:D1:82:1B:B4:AF:AC:F0:AA:9A:58:B5:D5:7A:33:8A:3A:FB:C
Alias name: mozillacert150.pem
   SHA1: 33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
   SHA256:
 EF:3C:B4:17:FC:8E:BF:6F:97:87:6C:9E:4E:CE:39:DE:1E:A5:FE:64:91:41:D1:02:8B:7D:11:C0:B2:29:8C:E
Alias name: mozillacert151.pem
   SHA1: AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
   SHA256:
 7F:12:CD:5F:7E:5E:29:0E:C7:D8:51:79:D5:B7:2C:20:A5:BE:75:08:FF:DB:5B:F8:1A:B9:68:4A:7F:C9:F6:6
Alias name: mozillacert16.pem
   SHA1: DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
   SHA256:
 06:87:26:03:31:A7:24:03:D9:09:F1:05:E6:9B:CF:0D:32:E1:BD:24:93:FF:C6:D9:20:6D:11:BC:D6:77:07:3
Alias name: mozillacert17.pem
   SHA1: 40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
   SHA256:
 76:7C:95:5A:76:41:2C:89:AF:68:8E:90:A1:C7:0F:55:6C:FD:6B:60:25:DB:EA:10:41:6D:7E:B6:83:1F:8C:4
Alias name: mozillacert18.pem
   SHA1: 79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
   SHA256:
 44:04:E3:3B:5E:14:0D:CF:99:80:51:FD:FC:80:28:C7:C8:16:15:C5:EE:73:7B:11:1B:58:82:33:A9:B5:35:A
Alias name: mozillacert19.pem
   SHA1: B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
   SHA256:
 C4:70:CF:54:7E:23:02:B9:77:FB:29:DD:71:A8:9A:7B:6C:1F:60:77:7B:03:29:F5:60:17:F3:28:BF:4F:6B:E
Alias name: mozillacert2.pem
   SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
   SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
```

```
Alias name: mozillacert20.pem
   SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
   SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: mozillacert21.pem
   SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
   SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: mozillacert22.pem
   SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
   SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: mozillacert23.pem
   SHA1: 91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
   SHA256:
 8D:72:2F:81:A9:C1:13:C0:79:1D:F1:36:A2:96:6D:B2:6C:95:0A:97:1D:B4:6B:41:99:F4:EA:54:B7:8B:FB:9
Alias name: mozillacert24.pem
   SHA1: 59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
   SHA256:
 66:8C:83:94:7D:A6:3B:72:4B:EC:E1:74:3C:31:A0:E6:AE:D0:DB:8E:C5:B3:1B:E3:77:BB:78:4F:91:B6:71:6
Alias name: mozillacert25.pem
   SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
   SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: mozillacert26.pem
   SHA1: 87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
   SHA256:
 F1:C1:B5:0A:E5:A2:0D:D8:03:0E:C9:F6:BC:24:82:3D:D3:67:B5:25:57:59:B4:E7:1B:61:FC:E9:F7:37:5D:7
Alias name: mozillacert27.pem
   SHA1: 3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
   SHA256:
 42:00:F5:04:3A:C8:59:0E:BB:52:7D:20:9E:D1:50:30:29:FB:CB:D4:1C:A1:B5:06:EC:27:F1:5A:DE:7D:AC:6
Alias name: mozillacert28.pem
   SHA1: 66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
   SHA256:
 0C:2C:D6:3D:F7:80:6F:A3:99:ED:E8:09:11:6B:57:5B:F8:79:89:F0:65:18:F9:80:8C:86:05:03:17:8B:AF:6
Alias name: mozillacert29.pem
   SHA1: 74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
   SHA256:
 15:F0:BA:00:A3:AC:7A:F3:AC:88:4C:07:2B:10:11:A0:77:BD:77:C0:97:F4:01:64:B2:F8:59:8A:BD:83:86:0
Alias name: mozillacert3.pem
   SHA1: 87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
   SHA256:
 39:DF:7B:68:2B:7B:93:8F:84:71:54:81:CC:DE:8D:60:D8:F2:2E:C5:98:87:7D:0A:AA:C1:2B:59:18:2B:03:1
```

```
Alias name: mozillacert30.pem
  SHA1: E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
  SHA256:
 A7:12:72:AE:AA:A3:CF:E8:72:7F:7F:B3:9F:0F:B3:D1:E5:42:6E:90:60:B0:6E:E6:F1:3E:9A:3C:58:33:CD:4
Alias name: mozillacert31.pem
  SHA1: 9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
  SHA256:
 17:93:92:7A:06:14:54:97:89:AD:CE:2F:8F:34:F7:F0:B6:6D:0F:3A:E3:A3:B8:4D:21:EC:15:DB:BA:4F:AD:C
Alias name: mozillacert32.pem
  SHA1: 60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
  SHA256:
 B9:BE:A7:86:0A:96:2E:A3:61:1D:AB:97:AB:6D:A3:E2:1C:10:68:B9:7D:55:57:5E:D0:E1:12:79:C1:1C:89:3
Alias name: mozillacert33.pem
  SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
  SHA256:
 A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:3
Alias name: mozillacert34.pem
  SHA1: 59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
  SHA256:
 41:C9:23:86:6A:B4:CA:D6:B7:AD:57:80:81:58:2E:02:07:97:A6:CB:DF:4F:FF:78:CE:83:96:B3:89:37:D7:F
Alias name: mozillacert35.pem
  SHA1: 2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
  SHA256:
 92:BF:51:19:AB:EC:CA:D0:B1:33:2D:C4:E1:D0:5F:BA:75:B5:67:90:44:EE:0C:A2:6E:93:1F:74:4F:2F:33:C
Alias name: mozillacert36.pem
  SHA1: 23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
  SHA256:
 32:7A:3D:76:1A:BA:DE:A0:34:EB:99:84:06:27:5C:B1:A4:77:6E:FD:AE:2F:DF:6D:01:68:EA:1C:4F:55:67:D
Alias name: mozillacert37.pem
  SHA1: B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
  SHA256:
 E3:B6:A2:DB:2E:D7:CE:48:84:2F:7A:C5:32:41:C7:B7:1D:54:14:4B:FB:40:C1:1F:3F:1D:0B:42:F5:EE:A1:2
Alias name: mozillacert38.pem
  SHA1: CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
  SHA256:
 A6:C5:1E:0D:A5:CA:0A:93:09:D2:E4:C0:E4:0C:2A:F9:10:7A:AE:82:03:85:7F:E1:98:E3:E7:69:E3:43:08:5
Alias name: mozillacert39.pem
  SHA1: AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
  SHA256:
 E6:B8:F8:76:64:85:F8:07:AE:7F:8D:AC:16:70:46:1F:07:C0:A1:3E:EF:3A:1F:F7:17:53:8D:7A:BA:D3:91:B
Alias name: mozillacert4.pem
  SHA1: E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
  SHA256:
 0B:5E:ED:4E:84:64:03:CF:55:E0:65:84:84:40:ED:2A:82:75:8B:F5:B9:AA:1F:25:3D:46:13:CF:A0:80:FF:3
```

```
Alias name: mozillacert40.pem
  SHA1: 80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
  SHA256:
 8D:A0:84:FC:F9:9C:E0:77:22:F8:9B:32:05:93:98:06:FA:5C:B8:11:E1:C8:13:F6:A1:08:C7:D3:36:B3:40:8
Alias name: mozillacert41.pem
  SHA1: 6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
  SHA256:
 EB:F3:C0:2A:87:89:B1:FB:7D:51:19:95:D6:63:B7:29:06:D9:13:CE:0D:5E:10:56:8A:8A:77:E2:58:61:67:E
Alias name: mozillacert42.pem
  SHA1: 85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
  SHA256:
 B6:19:1A:50:D0:C3:97:7F:7D:A9:9B:CD:AA:C8:6A:22:7D:AE:B9:67:9E:C7:0B:A3:B0:C9:D9:22:71:C1:70:D
Alias name: mozillacert43.pem
  SHA1: F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
  SHA256:
 50:79:41:C7:44:60:A0:B4:70:86:22:0D:4E:99:32:57:2A:B5:D1:B5:BB:CB:89:80:AB:1C:B1:76:51:A8:44:D
Alias name: mozillacert44.pem
  SHA1: 5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
  SHA256:
 96:0A:DF:00:63:E9:63:56:75:0C:29:65:DD:0A:08:67:DA:0B:9C:BD:6E:77:71:4A:EA:FB:23:49:AB:39:3D:A
Alias name: mozillacert45.pem
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
  SHA256:
 C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
Alias name: mozillacert46.pem
  SHA1: 40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
  SHA256:
 EC:C3:E9:C3:40:75:03:BE:E0:91:AA:95:2F:41:34:8F:F8:8B:AA:86:3B:22:64:BE:FA:C8:07:90:15:74:E9:3
Alias name: mozillacert47.pem
  SHA1: 1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
  SHA256:
 E4:C7:34:30:D7:A5:B5:09:25:DF:43:37:0A:0D:21:6E:9A:79:B9:D6:DB:83:73:A0:C6:9E:B1:CC:31:C7:C5:2
Alias name: mozillacert48.pem
  SHA1: A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
  SHA256:
 0F:4E:9C:DD:26:4B:02:55:50:D1:70:80:63:40:21:4F:E9:44:34:C9:B0:2F:69:7E:C7:10:FC:5F:EA:FB:5E:3
Alias name: mozillacert49.pem
  SHA1: 61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
  SHA256:
 B7:B1:2B:17:1F:82:1D:AA:99:0C:D0:FE:50:87:B1:28:44:8B:A8:E5:18:4F:84:C5:1E:02:B5:C8:FB:96:2B:2
Alias name: mozillacert5.pem
  SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
  SHA256:
 CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
```

```
Alias name: mozillacert50.pem
  SHA1: 8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
  SHA256:
 35:AE:5B:DD:D8:F7:AE:63:5C:FF:BA:56:82:A8:F0:0B:95:F4:84:62:C7:10:8E:E9:A0:E5:29:2B:07:4A:AF:B
Alias name: mozillacert51.pem
  SHA1: FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
  SHA256:
 EA:A9:62:C4:FA:4A:6B:AF:EB:E4:15:19:6D:35:1C:CD:88:8D:4F:53:F3:FA:8A:E6:D7:C4:66:A9:4E:60:42:B
Alias name: mozillacert52.pem
  SHA1: 8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
  SHA256:
 E2:83:93:77:3D:A8:45:A6:79:F2:08:0C:C7:FB:44:A3:B7:A1:C3:79:2C:B7:EB:77:29:FD:CB:6A:8D:99:AE:A
Alias name: mozillacert53.pem
  SHA1: 7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
  SHA256:
 2D:47:43:7D:E1:79:51:21:5A:12:F3:C5:8E:51:C7:29:A5:80:26:EF:1F:CC:0A:5F:B3:D9:DC:01:2F:60:0D:1
Alias name: mozillacert54.pem
  SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
  SHA256:
 B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
Alias name: mozillacert55.pem
  SHA1: AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
  SHA256:
 A4:31:0D:50:AF:18:A6:44:71:90:37:2A:86:AF:AF:8B:95:1F:FB:43:1D:83:7F:1E:56:88:B4:59:71:ED:15:5
Alias name: mozillacert56.pem
  SHA1: F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
  SHA256:
 4B:03:F4:58:07:AD:70:F2:1B:FC:2C:AE:71:C9:FD:E4:60:4C:06:4C:F5:FF:B6:86:BA:E5:DB:AA:D7:FD:D3:4
Alias name: mozillacert57.pem
  SHA1: D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
  SHA256:
 F9:E6:7D:33:6C:51:00:2A:C0:54:C6:32:02:2D:66:DD:A2:E7:E3:FF:F1:0A:D0:61:ED:31:D8:BB:B4:10:CF:B
Alias name: mozillacert58.pem
  SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
  SHA256:
 5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
Alias name: mozillacert59.pem
  SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
  SHA256:
 23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
Alias name: mozillacert6.pem
  SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
  SHA256:
 C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
```

```
Alias name: mozillacert60.pem
  SHA1: 3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
  SHA256:
 BF:0F:EE:FB:9E:3A:58:1A:D5:F9:E9:DB:75:89:98:57:43:D2:61:08:5C:4D:31:4F:6F:5D:72:59:AA:42:16:1
Alias name: mozillacert61.pem
  SHA1: E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
  SHA256:
 03:95:0F:B4:9A:53:1F:3E:19:91:94:23:98:DF:A9:E0:EA:32:D7:BA:1C:DD:9B:C8:5D:B5:7E:D9:40:0B:43:4
Alias name: mozillacert62.pem
  SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
  SHA256:
 A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
Alias name: mozillacert63.pem
  SHA1: 89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
  SHA256:
 3C:5F:81:FE:A5:FA:B8:2C:64:BF:A2:EA:EC:AF:CD:E8:E0:77:FC:86:20:A7:CA:E5:37:16:3D:F3:6E:DB:F3:7
Alias name: mozillacert64.pem
  SHA1: 62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
  SHA256:
 AB:70:36:36:5C:71:54:AA:29:C2:C2:9F:5D:41:91:16:3B:16:2A:22:25:01:13:57:D5:6D:07:FF:A7:BC:1F:7
Alias name: mozillacert65.pem
  SHA1: 69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
  SHA256:
 BC:23:F9:8A:31:3C:B9:2D:E3:BB:FC:3A:5A:9F:44:61:AC:39:49:4C:4A:E1:5A:9E:9D:F1:31:E9:9B:73:01:9
Alias name: mozillacert66.pem
  SHA1: DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
  SHA256:
 E6:09:07:84:65:A4:19:78:0C:B6:AC:4C:1C:0B:FB:46:53:D9:D9:CC:6E:B3:94:6E:B7:F3:D6:99:97:BA:D5:9
Alias name: mozillacert67.pem
  SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
  SHA256:
 CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
Alias name: mozillacert68.pem
  SHA1: AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
  SHA256:
 04:04:80:28:BF:1F:28:64:D4:8F:9A:D4:D8:32:94:36:6A:82:88:56:55:3F:3B:14:30:3F:90:14:7F:5D:40:E
Alias name: mozillacert69.pem
  SHA1: 2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
  SHA256:
 25:30:CC:8E:98:32:15:02:BA:D9:6F:9B:1F:BA:1B:09:9E:2D:29:9E:0F:45:48:BB:91:4F:36:3B:C0:D4:53:1
Alias name: mozillacert7.pem
  SHA1: AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
  SHA256:
 14:65:FA:20:53:97:B8:76:FA:A6:F0:A9:95:8E:55:90:E4:0F:CC:7F:AA:4F:B7:C2:C8:67:75:21:FB:5F:B6:5
```

```
Alias name: mozillacert70.pem
  SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
  SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: mozillacert71.pem
  SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
  SHA256:
 13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
Alias name: mozillacert72.pem
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: mozillacert73.pem
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: mozillacert74.pem
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: mozillacert75.pem
  SHA1: D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
  SHA256:
 08:29:7A:40:47:DB:A2:36:80:C7:31:DB:6E:31:76:53:CA:78:48:E1:BE:BD:3A:0B:01:79:A7:07:F9:2C:F1:7
Alias name: mozillacert76.pem
  SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
  SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
Alias name: mozillacert77.pem
  SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
  SHA256:
 EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
Alias name: mozillacert78.pem
  SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
  SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: mozillacert79.pem
  SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
  SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: mozillacert8.pem
  SHA1: 3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
  SHA256:
 C7:66:A9:BE:F2:D4:07:1C:86:3A:31:AA:49:20:E8:13:B2:D1:98:60:8C:B7:B7:CF:E2:11:43:B8:36:DF:09:E
```

```
Alias name: mozillacert80.pem
   SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
   SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: mozillacert81.pem
   SHA1: 07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
   SHA256:
 5C:58:46:8D:55:F5:8E:49:7E:74:39:82:D2:B5:00:10:B6:D1:65:37:4A:CF:83:A7:D4:A3:2D:B7:68:C4:40:8
Alias name: mozillacert82.pem
   SHA1: 2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
   SHA256:
 FC:BF:E2:88:62:06:F7:2B:27:59:3C:8B:07:02:97:E1:2D:76:9E:D1:0E:D7:93:07:05:A8:09:8E:FF:C1:4D:1
Alias name: mozillacert83.pem
   SHA1: A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
   SHA256:
 8C:4E:DF:D0:43:48:F3:22:96:9E:7E:29:A4:CD:4D:CA:00:46:55:06:1C:16:E1:B0:76:42:2E:F3:42:AD:63:0
Alias name: mozillacert84.pem
   SHA1: D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
   SHA256:
 79:3C:BF:45:59:B9:FD:E3:8A:B2:2D:F1:68:69:F6:98:81:AE:14:C4:B0:13:9A:C7:88:A7:8A:1A:FC:CA:02:F
Alias name: mozillacert85.pem
   SHA1: CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
   SHA256:
 BF:D8:8F:E1:10:1C:41:AE:3E:80:1B:F8:BE:56:35:0E:E9:BA:D1:A6:B9:BD:51:5E:DC:5C:6D:5B:87:11:AC:4
Alias name: mozillacert86.pem
   SHA1: 74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
   SHA256:
 E7:68:56:34:EF:AC:F6:9A:CE:93:9A:6B:25:5B:7B:4F:AB:EF:42:93:5B:50:A2:65:AC:B5:CB:60:27:E4:4E:7
Alias name: mozillacert87.pem
   SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
   SHA256:
 51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: mozillacert88.pem
   SHA1: FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
   SHA256:
 BC:10:4F:15:A4:8B:E7:09:DC:A5:42:A7:E1:D4:B9:DF:6F:05:45:27:E8:02:EA:A9:2D:59:54:44:25:8A:FE:7
Alias name: mozillacert89.pem
   SHA1: C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
   SHA256:
 E3:89:36:0D:0F:DB:AE:B3:D2:50:58:4B:47:30:31:4E:22:2F:39:C1:56:A0:20:14:4E:8D:96:05:61:79:15:0
Alias name: mozillacert9.pem
   SHA1: F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
   SHA256:
 76:00:29:5E:EF:E8:5B:9E:1F:D6:24:DB:76:06:2A:AA:AE:59:81:8A:54:D2:77:4C:D4:C0:B2:C0:11:31:E1:B
```

```
Alias name: mozillacert90.pem
   SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
   SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: mozillacert91.pem
   SHA1: 3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
   SHA256:
 C1:B4:82:99:AB:A5:20:8F:E9:63:0A:CE:55:CA:68:A0:3E:DA:5A:51:9C:88:02:A0:D3:A6:73:BE:8F:8E:55:7
Alias name: mozillacert92.pem
   SHA1: A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
   SHA256:
 E1:78:90:EE:09:A3:FB:F4:F4:8B:9C:41:4A:17:D6:37:B7:A5:06:47:E9:BC:75:23:22:72:7F:CC:17:42:A9:1
Alias name: mozillacert93.pem
   SHA1: 31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
   SHA256:
 C7:BA:65:67:DE:93:A7:98:AE:1F:AA:79:1E:71:2D:37:8F:AE:1F:93:C4:39:7F:EA:44:1B:B7:CB:E6:FD:59:9
Alias name: mozillacert94.pem
   SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
   SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
Alias name: mozillacert95.pem
   SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
   SHA256:
 ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: mozillacert96.pem
   SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
   SHA256:
 FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: mozillacert97.pem
   SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
   SHA256:
 83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: mozillacert98.pem
   SHA1: C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
   SHA256:
 3E:84:BA:43:42:90:85:16:E7:75:73:C0:99:2F:09:79:CA:08:4E:46:85:68:1F:F1:95:CC:BA:8A:22:9B:8A:7
Alias name: mozillacert99.pem
   SHA1: F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
   SHA256:
 97:8C:D9:66:F2:FA:A0:7B:A7:AA:95:00:D9:C0:2E:9D:77:F2:CD:AD:A6:AD:6B:A7:4A:F4:B9:1C:66:59:3C:5
Alias name: netlockaranyclassgoldfotanusitvany
   SHA1: 06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
   SHA256:
 6C:61:DA:C3:A2:DE:F0:31:50:6B:E0:36:D2:A6:FE:40:19:94:FB:D1:3D:F9:C8:D4:66:59:92:74:C4:46:EC:9
```

```
Alias name: networksolutionscertificateauthority
  SHA1: 74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
  SHA256:
 15:F0:BA:00:A3:AC:7A:F3:AC:88:4C:07:2B:10:11:A0:77:BD:77:C0:97:F4:01:64:B2:F8:59:8A:BD:83:86:0
Alias name: oistewisekeyglobalrootgaca
  SHA1: 59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
  SHA256:
 41:C9:23:86:6A:B4:CA:D6:B7:AD:57:80:81:58:2E:02:07:97:A6:CB:DF:4F:FF:78:CE:83:96:B3:89:37:D7:F
Alias name: oistewisekeyglobalrootgbca
  SHA1: 0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
  SHA256:
 6B:9C:08:E8:6E:B0:F7:67:CF:AD:65:CD:98:B6:21:49:E5:49:4A:67:F5:84:5E:7B:D1:ED:01:9F:27:B8:6B:D
Alias name: oistewisekeyglobalrootgcca
  SHA1: E0:11:84:5E:34:DE:BE:88:81:B9:9C:F6:16:26:D1:96:1F:C3:B9:31
  SHA256:
 85:60:F9:1C:36:24:DA:BA:95:70:B5:FE:A0:DB:E3:6F:F1:1A:83:23:BE:94:86:85:4F:B3:F3:4A:55:71:19:8
Alias name: quovadisrootca
  SHA1: DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
  SHA256:
 A4:5E:DE:3B:BB:F0:9C:8A:E1:5C:72:EF:C0:72:68:D6:93:A2:1C:99:6F:D5:1E:67:CA:07:94:60:FD:6D:88:7
Alias name: quovadisrootca1g3
  SHA1: 1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
  SHA256:
 8A:86:6F:D1:B2:76:B5:7E:57:8E:92:1C:65:82:8A:2B:ED:58:E9:F2:F2:88:05:41:34:B7:F1:F4:BF:C9:CC:7
Alias name: quovadisrootca2
  SHA1: CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
  SHA256:
 85:A0:DD:7D:D7:20:AD:B7:FF:05:F8:3D:54:2B:20:9D:C7:FF:45:28:F7:D6:77:B1:83:89:FE:A5:E5:C4:9E:8
Alias name: quovadisrootca2g3
  SHA1: 09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
  SHA256:
 8F:E4:FB:0A:F9:3A:4D:0D:67:DB:0B:EB:B2:3E:37:C7:1B:F3:25:DC:BC:DD:24:0E:A0:4D:AF:58:B4:7E:18:4
Alias name: quovadisrootca3
  SHA1: 1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
  SHA256:
 18:F1:FC:7F:20:5D:F8:AD:DD:EB:7F:E0:07:DD:57:E3:AF:37:5A:9C:4D:8D:73:54:6B:F4:F1:FE:D1:E1:8D:3
Alias name: quovadisrootca3g3
  SHA1: 48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
  SHA256:
 88:EF:81:DE:20:2E:B0:18:45:2E:43:F8:64:72:5C:EA:5F:BD:1F:C2:D9:D2:05:73:07:09:C5:D8:B8:69:0F:4
Alias name: secomevrootca1
  SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
  SHA256:
 A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:3
```

```
Alias name: secomscrootca1
    SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
    SHA256:
  E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: secomscrootca2
    SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
    SHA256:
  51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: secomvalicertclass1ca
    SHA1: E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
    SHA256:
  F4:C1:49:55:1A:30:13:A3:5B:C7:BF:FE:17:A7:F3:44:9B:C1:AB:5B:5A:0A:E7:4B:06:C2:3B:90:00:4C:01:0
Alias name: secureglobalca
    SHA1: 3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
    SHA256:
  42:00:F5:04:3A:C8:59:0E:BB:52:7D:20:9E:D1:50:30:29:FB:CB:D4:1C:A1:B5:06:EC:27:F1:5A:DE:7D:AC:6
Alias name: securesignrootca11
    SHA1: 3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
    SHA256:
  BF:0F:EE:FB:9E:3A:58:1A:D5:F9:E9:DB:75:89:98:57:43:D2:61:08:5C:4D:31:4F:6F:5D:72:59:AA:42:16:1
Alias name: securetrustca
    SHA1: 87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
    SHA256:
  F1:C1:B5:0A:E5:A2:0D:D8:03:0E:C9:F6:BC:24:82:3D:D3:67:B5:25:57:59:B4:E7:1B:61:FC:E9:F7:37:5D:7
Alias name: securitycommunicationrootca
    SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
    SHA256:
  E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: securitycommunicationrootca2
    SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
    SHA256:
  51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: soneraclass1ca
    SHA1: 07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
    SHA256:
  CD:80:82:84:CF:74:6F:F2:FD:6E:B5:8A:A1:D5:9C:4A:D4:B3:CA:56:FD:C6:27:4A:89:26:A7:83:5F:32:31:3
Alias name: soneraclass2ca
    SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
    SHA256:
  79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
Alias name: soneraclass2rootca
    SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
    SHA256:
  79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
```

```
Alias name: sslcomevrootcertificationauthorityecc
  SHA1: 4C:DD:51:A3:D1:F5:20:32:14:B0:C6:C5:32:23:03:91:C7:46:42:6D
  SHA256:
 22:A2:C1:F7:BD:ED:70:4C:C1:E7:01:B5:F4:08:C3:10:88:0F:E9:56:B5:DE:2A:4A:44:F9:9C:87:3A:25:A7:C
Alias name: sslcomevrootcertificationauthorityrsar2
  SHA1: 74:3A:F0:52:9B:D0:32:A0:F4:4A:83:CD:D4:BA:A9:7B:7C:2E:C4:9A
  SHA256:
 2E:7B:F1:6C:C2:24:85:A7:BB:E2:AA:86:96:75:07:61:B0:AE:39:BE:3B:2F:E9:D0:CC:6D:4E:F7:34:91:42:5
Alias name: sslcomrootcertificationauthorityecc
  SHA1: C3:19:7C:39:24:E6:54:AF:1B:C4:AB:20:95:7A:E2:C3:0E:13:02:6A
  SHA256:
 34:17:BB:06:CC:60:07:DA:1B:96:1C:92:0B:8A:B4:CE:3F:AD:82:0E:4A:A3:0B:9A:CB:C4:A7:4E:BD:CE:BC:6
Alias name: sslcomrootcertificationauthorityrsa
  SHA1: B7:AB:33:08:D1:EA:44:77:BA:14:80:12:5A:6F:BD:A9:36:49:0C:BB
  SHA256:
 85:66:6A:56:2E:E0:BE:5C:E9:25:C1:D8:89:0A:6F:76:A8:7E:C1:6D:4D:7D:5F:29:EA:74:19:CF:20:12:3B:6
Alias name: staatdernederlandenevrootca
  SHA1: 76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
  SHA256:
 4D:24:91:41:4C:FE:95:67:46:EC:4C:EF:A6:CF:6F:72:E2:8A:13:29:43:2F:9D:8A:90:7A:C4:CB:5D:AD:C1:5
Alias name: staatdernederlandenrootcag3
  SHA1: D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
  SHA256:
 3C:4F:B0:B9:5A:B8:B3:00:32:F4:32:B8:6F:53:5F:E1:72:C1:85:D0:FD:39:86:58:37:CF:36:18:7F:A6:F4:2
Alias name: starfieldclass2ca
  SHA1: AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
  SHA256:
 14:65:FA:20:53:97:B8:76:FA:A6:F0:A9:95:8E:55:90:E4:0F:CC:7F:AA:4F:B7:C2:C8:67:75:21:FB:5F:B6:5
Alias name: starfieldrootcertificateauthorityg2
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: starfieldrootg2ca
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: starfieldservicesrootcertificateauthorityg2
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: starfieldservicesrootg2ca
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: swisssigngoldcag2
  SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
  SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: swisssigngoldg2ca
  SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
  SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: swisssignplatinumg2ca
  SHA1: 56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
  SHA256:
 3B:22:2E:56:67:11:E9:92:30:0D:C0:B1:5A:B9:47:3D:AF:DE:F8:C8:4D:0C:EF:7D:33:17:B4:C1:82:1D:14:3
Alias name: swisssignsilvercag2
  SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
  SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: swisssignsilverg2ca
  SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
  SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: szafirrootca2
  SHA1: E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
  SHA256:
 A1:33:9D:33:28:1A:0B:56:E5:57:D3:D3:2B:1C:E7:F9:36:7E:B0:94:BD:5F:A7:2A:7E:50:04:C8:DE:D7:CA:F
Alias name: teliasonerarootcav1
  SHA1: 43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
  SHA256:
 DD:69:36:FE:21:F8:F0:77:C1:23:A1:A5:21:C1:22:24:F7:22:55:B7:3E:03:A7:26:06:93:E8:A2:4B:0F:A3:8
Alias name: thawtepersonalfreemailca
  SHA1: E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
  SHA256:
 5B:38:BD:12:9E:83:D5:A0:CA:D2:39:21:08:94:90:D5:0D:4A:AE:37:04:28:F8:DD:FF:FF:FA:4C:15:64:E1:8
Alias name: thawtepremiumserverca
  SHA1: E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
  SHA256:
 3F:9F:27:D5:83:20:4B:9E:09:C8:A3:D2:06:6C:4B:57:D3:A2:47:9C:36:93:65:08:80:50:56:98:10:5D:BC:E
Alias name: thawteprimaryrootca
  SHA1: 91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
  SHA256:
 8D:72:2F:81:A9:C1:13:C0:79:1D:F1:36:A2:96:6D:B2:6C:95:0A:97:1D:B4:6B:41:99:F4:EA:54:B7:8B:FB:9
Alias name: thawteprimaryrootcag2
  SHA1: AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
  SHA256:
 A4:31:0D:50:AF:18:A6:44:71:90:37:2A:86:AF:AF:8B:95:1F:FB:43:1D:83:7F:1E:56:88:B4:59:71:ED:15:5
```

```
Alias name: thawteprimaryrootcag3
  SHA1: F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
  SHA256:
 4B:03:F4:58:07:AD:70:F2:1B:FC:2C:AE:71:C9:FD:E4:60:4C:06:4C:F5:FF:B6:86:BA:E5:DB:AA:D7:FD:D3:4
Alias name: thawteserverca
  SHA1: 9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
  SHA256:
 87:C6:78:BF:B8:B2:5F:38:F7:E9:7B:33:69:56:BB:CF:14:4B:BA:CA:A5:36:47:E6:1A:23:25:BC:10:55:31:6
Alias name: trustcenterclass2caii
  SHA1: AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
  SHA256:
 E6:B8:F8:76:64:85:F8:07:AE:7F:8D:AC:16:70:46:1F:07:C0:A1:3E:EF:3A:1F:F7:17:53:8D:7A:BA:D3:91:B
Alias name: trustcenterclass4caii
  SHA1: A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
  SHA256:
 32:66:96:7E:59:CD:68:00:8D:9D:D3:20:81:11:85:C7:04:20:5E:8D:95:FD:D8:4F:1C:7B:31:1E:67:04:FC:3
Alias name: trustcenteruniversalcai
  SHA1: 6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
  SHA256:
 EB:F3:C0:2A:87:89:B1:FB:7D:51:19:95:D6:63:B7:29:06:D9:13:CE:0D:5E:10:56:8A:8A:77:E2:58:61:67:E
Alias name: trustcoreca1
  SHA1: 58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
  SHA256:
 5A:88:5D:B1:9C:01:D9:12:C5:75:93:88:93:8C:AF:BB:DF:03:1A:B2:D4:8E:91:EE:15:58:9B:42:97:1D:03:9
Alias name: trustcorrootcertca1
  SHA1: FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
  SHA256:
 D4:0E:9C:86:CD:8F:E4:68:C1:77:69:59:F4:9E:A7:74:FA:54:86:84:B6:C4:06:F3:90:92:61:F4:DC:E2:57:5
Alias name: trustcorrootcertca2
  SHA1: B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
  SHA256:
 07:53:E9:40:37:8C:1B:D5:E3:83:6E:39:5D:AE:A5:CB:83:9E:50:46:F1:BD:0E:AE:19:51:CF:10:FE:C7:C9:6
Alias name: trustisfpsrootca
  SHA1: 3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
  SHA256:
 C1:B4:82:99:AB:A5:20:8F:E9:63:0A:CE:55:CA:68:A0:3E:DA:5A:51:9C:88:02:A0:D3:A6:73:BE:8F:8E:55:7
Alias name: ttelesecglobalrootclass2
  SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
  SHA256:
 91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
Alias name: ttelesecglobalrootclass2ca
  SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
  SHA256:
 91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
```

```
Alias name: ttelesecglobalrootclass3
  SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
  SHA256:
 FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: ttelesecglobalrootclass3ca
  SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
  SHA256:
 FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: tubitakkamusmsslkoksertifikasisurum1
  SHA1: 31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
  SHA256:
 46:ED:C3:68:90:46:D5:3A:45:3F:B3:10:4A:B8:0D:CA:EC:65:8B:26:60:EA:16:29:DD:7E:86:79:90:64:87:1
Alias name: twcaglobalrootca
  SHA1: 9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
  SHA256:
 59:76:90:07:F7:68:5D:0F:CD:50:87:2F:9F:95:D5:75:5A:5B:2B:45:7D:81:F3:69:2B:61:0A:98:67:2F:0E:1
Alias name: twcarootcertificationauthority
  SHA1: CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
  SHA256:
 BF:D8:8F:E1:10:1C:41:AE:3E:80:1B:F8:BE:56:35:0E:E9:BA:D1:A6:B9:BD:51:5E:DC:5C:6D:5B:87:11:AC:4
Alias name: ucaextendedvalidationroot
  SHA1: A3:A1:B0:6F:24:61:23:4A:E3:36:A5:C2:37:FC:A6:FF:DD:F0:D7:3A
  SHA256:
 D4:3A:F9:B3:54:73:75:5C:96:84:FC:06:D7:D8:CB:70:EE:5C:28:E7:73:FB:29:4E:B4:1E:E7:17:22:92:4D:2
Alias name: ucaglobalg2root
  SHA1: 28:F9:78:16:19:7A:FF:18:25:18:AA:44:FE:C1:A0:CE:5C:B6:4C:8A
  SHA256:
 9B:EA:11:C9:76:FE:01:47:64:C1:BE:56:A6:F9:14:B5:A5:60:31:7A:BD:99:88:39:33:82:E5:16:1A:A0:49:3
Alias name: usertrustecc
  SHA1: D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
  SHA256:
 4F:F4:60:D5:4B:9C:86:DA:BF:BC:FC:57:12:E0:40:0D:2B:ED:3F:BC:4D:4F:BD:AA:86:E0:6A:DC:D2:A9:AD:7
Alias name: usertrustecccertificationauthority
  SHA1: D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
  SHA256:
 4F:F4:60:D5:4B:9C:86:DA:BF:BC:FC:57:12:E0:40:0D:2B:ED:3F:BC:4D:4F:BD:AA:86:E0:6A:DC:D2:A9:AD:7
Alias name: usertrustrsa
  SHA1: 2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
  SHA256:
 E7:93:C9:B0:2F:D8:AA:13:E2:1C:31:22:8A:CC:B0:81:19:64:3B:74:9C:89:89:64:B1:74:6D:46:C3:D4:CB:D
Alias name: usertrustrsacertificationauthority
  SHA1: 2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
  SHA256:
 E7:93:C9:B0:2F:D8:AA:13:E2:1C:31:22:8A:CC:B0:81:19:64:3B:74:9C:89:89:64:B1:74:6D:46:C3:D4:CB:D
```

```
Alias name: utndatacorpsgcca
  SHA1: 58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
  SHA256:
 85:FB:2F:91:DD:12:27:5A:01:45:B6:36:53:4F:84:02:4A:D6:8B:69:B8:EE:88:68:4F:F7:11:37:58:05:B3:4
Alias name: utnuserfirstclientauthemailca
  SHA1: B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
  SHA256:
 43:F2:57:41:2D:44:0D:62:74:76:97:4F:87:7D:A8:F1:FC:24:44:56:5A:36:7A:E6:0E:DD:C2:7A:41:25:31:A
Alias name: utnuserfirsthardwareca
  SHA1: 04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
  SHA256:
 6E:A5:47:41:D0:04:66:7E:ED:1B:48:16:63:4A:A3:A7:9E:6E:4B:96:95:0F:82:79:DA:FC:8D:9B:D8:81:21:3
Alias name: utnuserfirstobjectca
  SHA1: E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
  SHA256:
 6F:FF:78:E4:00:A7:0C:11:01:1C:D8:59:77:C4:59:FB:5A:F9:6A:3D:F0:54:08:20:D0:F4:B8:60:78:75:E5:8
Alias name: valicertclass2ca
  SHA1: 31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
  SHA256:
 58:D0:17:27:9C:D4:DC:63:AB:DD:B1:96:A6:C9:90:6C:30:C4:E0:87:83:EA:E8:C1:60:99:54:D6:93:55:59:6
Alias name: verisignc1g1.pem
  SHA1: 90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
  SHA256:
 D1:7C:D8:EC:D5:86:B7:12:23:8A:48:2C:E4:6F:A5:29:39:70:74:2F:27:6D:8A:B6:A9:E4:6E:E0:28:8F:33:5
Alias name: verisignc1g2.pem
  SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
  SHA256:
 34:1D:E9:8B:13:92:AB:F7:F4:AB:90:A9:60:CF:25:D4:BD:6E:C6:5B:9A:51:CE:6E:D0:67:D0:0E:C7:CE:9B:7
Alias name: verisignc1g3.pem
  SHA1: 20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
  SHA256:
 CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:C0:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:6
Alias name: verisignc1g6.pem
  SHA1: 51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
  SHA256:
 9D:19:0B:2E:31:45:66:68:5B:E8:A8:89:E2:7A:A8:C7:D7:AE:1D:8A:AD:DB:A3:C1:EC:F9:D2:48:63:CD:34:B
Alias name: verisignc2g1.pem
  SHA1: 67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
  SHA256:
 BD:46:9F:F4:5F:AA:E7:C5:4C:CB:D6:9D:3F:3B:00:22:55:D9:B0:6B:10:B1:D0:FA:38:8B:F9:6B:91:8B:2C:E
Alias name: verisignc2g2.pem
  SHA1: B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
  SHA256:
 3A:43:E2:20:FE:7F:3E:A9:65:3D:1E:21:74:2E:AC:2B:75:C2:0F:D8:98:03:05:BC:50:2C:AF:8C:2D:9B:41:A
```

```
Alias name: verisignc2g3.pem
    SHA1: 61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
    SHA256:
 92:A9:D9:83:3F:E1:94:4D:B3:66:E8:BF:AE:7A:95:B6:48:0C:2D:6C:6C:2A:1B:E6:5D:42:36:B6:08:FC:A1:B
Alias name: verisignc2g6.pem
    SHA1: 40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
    SHA256:
 CB:62:7D:18:B5:8A:D5:6D:DE:33:1A:30:45:6B:C6:5C:60:1A:4E:9B:18:DE:DC:EA:08:E7:DA:AA:07:81:5F:F
Alias name: verisignc3g1.pem
    SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
    SHA256:
 A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
Alias name: verisignc3g2.pem
    SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
    SHA256:
 83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: verisignc3g3.pem
    SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
    SHA256:
 EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
Alias name: verisignc3g4.pem
    SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
    SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
Alias name: verisignc3g5.pem
    SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
    SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: verisignc4g2.pem
    SHA1: 0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
    SHA256:
 44:64:0A:0A:0E:4D:00:0F:BD:57:4D:2B:8A:07:BD:B4:D1:DF:ED:3B:45:BA:AB:A7:6F:78:57:78:C7:01:19:6
Alias name: verisignc4g3.pem
    SHA1: C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
    SHA256:
 E3:89:36:0D:0F:DB:AE:B3:D2:50:58:4B:47:30:31:4E:22:2F:39:C1:56:A0:20:14:4E:8D:96:05:61:79:15:0
Alias name: verisignclass1ca
    SHA1: CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
    SHA256:
 51:84:7C:8C:BD:2E:9A:72:C9:1E:29:2D:2A:E2:47:D7:DE:1E:3F:D2:70:54:7A:20:EF:7D:61:0F:38:B8:84:2
Alias name: verisignclass1g2ca
    SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
    SHA256:
 34:1D:E9:8B:13:92:AB:F7:F4:AB:90:A9:60:CF:25:D4:BD:6E:C6:5B:9A:51:CE:6E:D0:67:D0:0E:C7:CE:9B:7
```

```
Alias name: verisignclass1g3ca
  SHA1: 20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
  SHA256:
 CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:C0:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:6
Alias name: verisignclass2g2ca
  SHA1: B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
  SHA256:
 3A:43:E2:20:FE:7F:3E:A9:65:3D:1E:21:74:2E:AC:2B:75:C2:0F:D8:98:03:05:BC:50:2C:AF:8C:2D:9B:41:A
Alias name: verisignclass2g3ca
  SHA1: 61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
  SHA256:
 92:A9:D9:83:3F:E1:94:4D:B3:66:E8:BF:AE:7A:95:B6:48:0C:2D:6C:6C:2A:1B:E6:5D:42:36:B6:08:FC:A1:B
Alias name: verisignclass3ca
  SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
  SHA256:
 A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
Alias name: verisignclass3g2ca
  SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
  SHA256:
 83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: verisignclass3g3ca
  SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
  SHA256:
 EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
Alias name: verisignclass3g4ca
  SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
  SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
Alias name: verisignclass3g5ca
  SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
  SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: verisignclass3publicprimarycertificationauthorityg4
  SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
  SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
Alias name: verisignclass3publicprimarycertificationauthorityg5
  SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
  SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: verisignroot.pem
  SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
  SHA256:
 23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
```

```
Alias name: verisigntsaca
  SHA1: 20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
  SHA256:
 CB:6B:05:D9:E8:E5:7C:D8:82:B1:0B:4D:B7:0D:E4:BB:1D:E4:2B:A4:8A:7B:D0:31:8B:63:5B:F6:E7:78:1A:9
Alias name: verisignuniversalrootca
  SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
  SHA256:
 23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
Alias name: verisignuniversalrootcertificationauthority
  SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
  SHA256:
 23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
Alias name: xrampglobalca
  SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
  SHA256:
 CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
Alias name: xrampglobalcaroot
  SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
  SHA256:
 CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
```

Using AWS WAF to protect your APIs

AWS WAF is a web application firewall that helps protect web applications and APIs from attacks. It enables you to configure a set of rules called a web access control list (web ACL) that allow, block, or count web requests based on customizable web security rules and conditions that you define. For more information, see How AWS WAF Works.

You can use AWS WAF to protect your API Gateway REST API from common web exploits, such as SQL injection and cross-site scripting (XSS) attacks. These could affect API availability and performance, compromise security, or consume excessive resources. For example, you can create rules to allow or block requests from specified IP address ranges, requests from CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, or requests that contain malicious script.

You can also create rules that match a specified string or a regular expression pattern in HTTP headers, method, query string, URI, and the request body (limited to the first 8 KB). Additionally, you can create rules to block attacks from specific user agents, bad bots, and content scrapers. For example, you can use rate-based rules to specify the number of web requests that are allowed by each client IP in a trailing, continuously updated, 5-minute period.

⚠ **Important**

AWS WAF is your first line of defense against web exploits. When AWS WAF is enabled on an API, AWS WAF rules are evaluated before other access control features, such as resource policies, IAM policies, Lambda authorizers, and Amazon Cognito authorizers. For example, if AWS WAF blocks access from a CIDR block that a resource policy allows, AWS WAF takes precedence and the resource policy isn't evaluated.

To enable AWS WAF for your API, you need to do the following:

1. Use the AWS WAF console, AWS SDK, or CLI to create a web ACL that contains the desired combination of AWS WAF managed rules and your own custom rules. For more information, see Getting Started with AWS WAF and Web access control lists (web ACLs).

   ⚠ **Important**

   API Gateway requires an AWS WAFV2 web ACL for a Regional application or an AWS WAF Classic Regional web ACL.

2. Associate the AWS WAF web ACL with an API stage. You can do this by using the AWS WAF console, AWS SDK, CLI, or by using the API Gateway console.

To associate an AWS WAF web ACL with an API Gateway API stage using the API Gateway console

To use the API Gateway console to associate an AWS WAF web ACL with an existing API Gateway API stage, use the following steps:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an existing API or create a new one.
3. In the main navigation pane, choose **Stages**, and then choose a stage.
4. In the **Stage details** section, choose **Edit**.
5. Under **Web application firewall (AWS WAF)**, select your web ACL.

If you are using AWS WAFV2, select an AWS WAFV2 web ACL for a Regional application. The web ACL and any other AWS WAFV2 resources that it uses must be located in the same Region as your API.

If you are using AWS WAF Classic Regional, select a Regional web ACL.

6. Choose **Save changes**.

Associate an AWS WAF web ACL with an API Gateway API stage using the AWS CLI

To use the AWS CLI to associate an AWS WAFV2 web ACL for a Regional application with an existing API Gateway API stage, call the associate-web-acl command, as in the following example:

```
aws wafv2 associate-web-acl \
--web-acl-arn  arn:aws:wafv2:{region}:111122223333:regional/webacl/test-cli/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
--resource-arn arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod
```

To use the AWS CLI to associate an AWS WAF Classic Regional web ACL with an existing API Gateway API stage, call the associate-web-acl command, as in the following example:

```
aws waf-regional associate-web-acl \
--web-acl-id 'aabc123a-fb4f-4fc6-becb-2b00831cadcf' \
--resource-arn 'arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
```

Associate an AWS WAF web ACL with an API stage using the AWS WAF REST API

To use the AWS WAFV2 REST API to associate an AWS WAFV2 web ACL for a Regional application with an existing API Gateway API stage, use the AssociateWebACL command, as in the following example:

```
import boto3

wafv2 = boto3.client('wafv2')

wafv2.associate_web_acl(
    WebACLArn='arn:aws:wafv2:{region}:111122223333:regional/webacl/test/abc6aa3b-
fc33-4841-b3db-0ef3d3825b25',
    ResourceArn='arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
)
```

To use the AWS WAF REST API to associate an AWS WAF Classic Regional web ACL with an existing API Gateway API stage, use the AssociateWebACL command, as in the following example:

```
import boto3

waf = boto3.client('waf-regional')

waf.associate_web_acl(
    WebACLId='aabc123a-fb4f-4fc6-becb-2b00831cadcf',
    ResourceArn='arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
)
```

Throttle API requests for better throughput

You can configure throttling and quotas for your APIs to help protect them from being overwhelmed by too many requests. Both throttles and quotas are applied on a best-effort basis and should be thought of as targets rather than guaranteed request ceilings.

API Gateway throttles requests to your API using the token bucket algorithm, where a token counts for a request. Specifically, API Gateway examines the rate and a burst of request submissions against all APIs in your account, per Region. In the token bucket algorithm, a burst can allow pre-defined overrun of those limits, but other factors can also cause limits to be overrun in some cases.

When request submissions exceed the steady-state request rate and burst limits, API Gateway begins to throttle requests. Clients may receive `429 Too Many Requests` error responses at this point. Upon catching such exceptions, the client can resubmit the failed requests in a way that is rate limiting.

As an API developer, you can set the target limits for individual API stages or methods to improve overall performance across all APIs in your account. Alternatively, you can enable usage plans to set throttles on client request submissions based on specified requests rates and quotas.

Topics

How throttling limit settings are applied in API Gateway

Before you configure throttle and quota settings for your API, it's useful to understand how they are applied by Amazon API Gateway.

Amazon API Gateway provides four basic types of throttling-related settings:

- *AWS throttling limits* are applied across all accounts and clients in a region. These limit settings exist to prevent your API—and your account—from being overwhelmed by too many requests. These limits are set by AWS and can't be changed by a customer.
- Per-account limits are applied to all APIs in an account in a specified Region. The account-level rate limit can be increased upon request - higher limits are possible with APIs that have shorter timeouts and smaller payloads. To request an increase of account-level throttling limits per Region, contact the AWS Support Center. For more information, see *Quotas and important notes*. Note that these limits can't be higher than the AWS throttling limits.
- Per-API, per-stage throttling limits are applied at the API method level for a stage. You can configure the same settings for all methods, or configure different throttle settings for each method. Note that these limits can't be higher than the AWS throttling limits.
- *Per-client throttling limits* are applied to clients that use API keys associated with your usage plan as client identifier. Note that these limits can't be higher than the per-account limits.

API Gateway throttling-related settings are applied in the following order:

1. Per-client or per-method throttling limits that you set for an API stage in a usage plan
2. Per-method throttling limits that you set for an API stage
3. Account-level throttling per Region
4. AWS Regional throttling

Account-level throttling per Region

By default, API Gateway limits the steady-state requests per second (RPS) across all APIs within an AWS account, per Region. It also limits the burst (that is, the maximum bucket size) across all APIs within an AWS account, per Region. In API Gateway, the burst limit represents the target maximum number of concurrent request submissions that API Gateway will fulfill before returning `429 Too Many Requests` error responses. For more information on throttling quotas, see *Quotas and important notes*.

Configuring API-level and stage-level throttling targets in a usage plan

In a usage plan, you can set a per-method throttling target for all methods at the API or stage level. You can specify a *throttling rate*, which is the rate, in requests per second, that tokens are added to the token bucket. You can also specify a *throttling burst*, which is the capacity of the token bucket.

You can use the AWS CLI, SDKs, and the AWS Management Console to create a usage plan. For more information about how to create a usage plan, see ???.

Configuring stage-level throttling targets

You can use the AWS CLI, SDKs, and the AWS Management Console to create stage-level throttling targets.

For more information about how to use the AWS Management Console to create stage-level throttling targets, see ???. For more information about how to use the AWS CLI to create stage-level throttling targets, see create-stage.

Configuring method-level throttling targets in a usage plan

You can set additional throttling targets at the method level in **Usage Plans** as shown in Create a usage plan. In the API Gateway console, these are set by specifying `Resource=<resource>`, `Method=<method>` in the **Configure Method Throttling** setting. For example, for the PetStore example, you might specify `Resource=/pets`, `Method=GET`.

Creating a private API in Amazon API Gateway

Using Amazon API Gateway, you can create private REST APIs that can only be accessed from your virtual private cloud in Amazon VPC by using an interface VPC endpoint. This is an endpoint network interface that you create in your VPC.

Using resource policies, you can allow or deny access to your API from selected VPCs and VPC endpoints, including across AWS accounts. Each endpoint can be used to access multiple private APIs. You can also use AWS Direct Connect to establish a connection from an on-premises network to Amazon VPC and access your private API over that connection.

⚠ **Important**

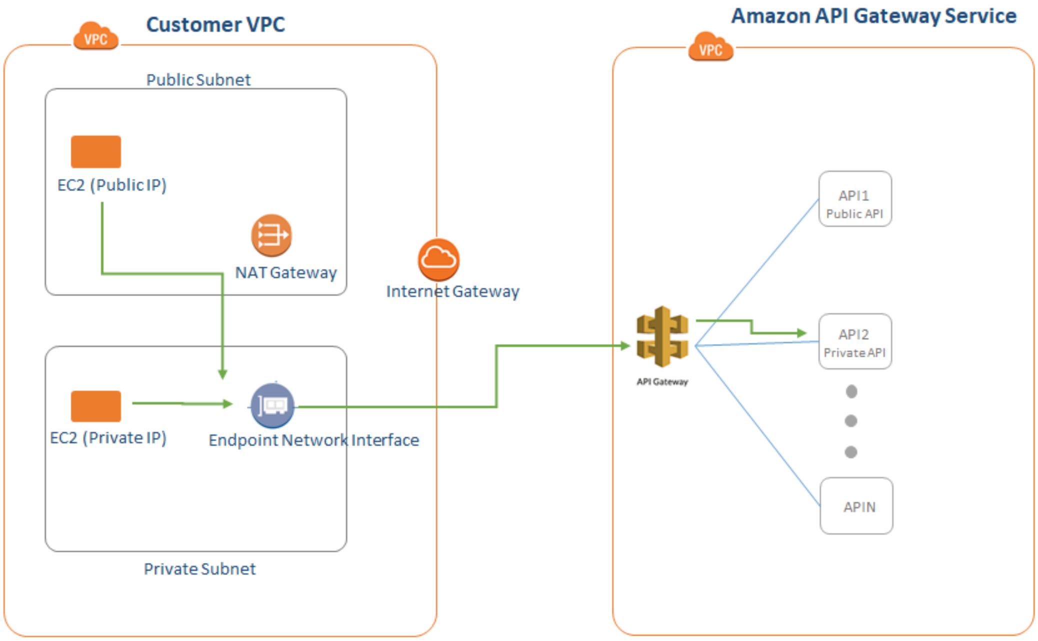
To restrict access to your private API to specific VPCs or VPC endpoints, add `aws:SourceVpc` or `aws:SourceVpce` conditions to your API's resource policy. For example policies, see the section called “Example: Allow private API traffic based on source VPC or VPC endpoint”.

In all cases, traffic to your private API uses secure connections and does not leave the Amazon network—it is isolated from the public internet.

You can access your private APIs through interface VPC endpoints for API Gateway as shown in the following diagram. If you have private DNS enabled, you can use private or public DNS names to access your APIs. If you have private DNS disabled, you can only use public DNS names.

Note

API Gateway private APIs only support TLS 1.2. Earlier TLS versions are not supported.



Private DNS Enabled: <api_id>.execute-api.<region>.amazonaws.com
OR
<vpce-id>.execute-api.<region>.vpce.amazonaws.com
Host: <api_id>.execute-api.<region>.amazonaws.com

Private DNS Disabled: <vpce-id>.execute-api.<region>.vpce.amazonaws.com
Host: <api_id>.execute-api.<region>.amazonaws.com

At a high level, the steps for creating a private API are as follows:

1. First, create an interface VPC endpoint for the API Gateway component service for API execution, known as `execute-api`, in your VPC.

2. Create and test your private API.

   a. Use one of the following procedures to create your API:

      - API Gateway console
      - API Gateway CLI
      - AWS SDK for JavaScript

   b. To grant access to your VPC endpoint, create a resource policy and attach it to your API.

   c. Test your API.

> **Note**
>
> The procedures below assume you already have a fully configured VPC. For more information, and to get started with creating a VPC, see Getting Started With Amazon VPC in the Amazon VPC User Guide.

Private API development considerations

- You can convert an existing public API (Regional or edge-optimized) to a private API, and you can convert a private API to a Regional API. You cannot convert a private API to an edge-optimized API. For more information, see ???.
- To grant access to your private API to VPCs and VPC endpoints, you need to create a resource policy and attach it to the newly created (or converted) API. Until you do so, all calls to the API will fail. For more information, see ???.
- Custom domain names are not supported for private APIs.
- You can use a single VPC endpoint to access multiple private APIs.
- You can associate or disassociate a VPC endpoint to a REST API, which gives a Route 53 alias DNS record and simplifies invoking your private API. For more information, see Associate or Disassociate a VPC Endpoint with a Private REST API.

> **Note**
>
> VPC endpoints for private APIs are subject to the same limitations as other interface VPC endpoints. For more information, see Interface Endpoint Properties and Limitations in the

AWS PrivateLink Guide. For more information about using API Gateway with shared VPCs and shared subnets, see Shared subnets in the *AWS PrivateLink Guide*.

Topics

Create an interface VPC endpoint for API Gateway `execute-api`

The API Gateway component service for API execution is called `execute-api`. To access your private API once it's deployed, you need to create an interface VPC endpoint for it in your VPC.

After you've created your VPC endpoint, you can use it to access multiple private APIs.

To create an interface VPC endpoint for API Gateway `execute-api`

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**, **Create Endpoint**.
3. For **Service category**, ensure that **AWS services** is selected.
4. For **Service Name**, choose the API Gateway service endpoint, including the AWS Region that you want to connect to. This is in the form `com.amazonaws.`*`region`*`.execute-api`—for example, `com.amazonaws.us-east-1.execute-api`.

   For **Type**, ensure that it indicates **Interface**.
5. Complete the following information:

   - For **VPC**, choose the VPC that you want to create the endpoint in.
   - For **Subnets**, choose the subnets (Availability Zones) in which to create the endpoint network interfaces. To improve the availability of your API, choose multiple subnets.

> **Note**
>
> Not all Availability Zones may be supported for all AWS services.

- For **Enable Private DNS Name**, leave the check box selected. Private DNS is enabled by default.

  When private DNS is enabled, you're able to access your API via private or public DNS. (This setting doesn't affect who can access your API, only which DNS addresses they can use.) However, you cannot access public APIs from a VPC by using an API Gateway VPC endpoint with private DNS enabled. Note that these DNS settings don't affect the ability to call these public APIs from the VPC if you're using an edge-optimized custom domain name to access the public API. Using an edge-optimized custom domain name to access your public API (while using private DNS to access your private API) is one way to access both public and private APIs from a VPC where the endpoint has been created with private DNS enabled.

  > **Note**
  >
  > Leaving private DNS enabled is the recommended choice. If you choose not to enable private DNS, you're only able to access your API via public DNS. To learn more, see How to invoke a private API.

  To use the private DNS option, the `enableDnsSupport` and `enableDnsHostnames` attributes of your VPC must be set to `true`. For more information, see DNS Support in Your VPC and Updating DNS Support for Your VPC in the Amazon VPC User Guide.

- For **Security group**, select the security group to associate with the VPC endpoint network interfaces.

  The security group you choose must be set to allow TCP Port 443 inbound HTTPS traffic from either an IP range in your VPC or another security group in your VPC.

6. Choose **Create endpoint**.

Create a private API using the API Gateway console

To create a private API using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Create API**.
3. Under **REST API**, choose **Build**.
4. For **Name**, enter a name.
5. (Optional) For **Description**, enter a description.
6. For **API endpoint type**, select **Private**.
7. Choose **Create API**.

From here on, you can set up API methods and their associated integrations as described in steps 1-6 of ???.

> **Note**
>
> Until your API has a resource policy that grants access to your VPC or VPC endpoint, all API calls will fail. Before you test and deploy your API, you need to create a resource policy and attach it to the API as described in ???.

Create a private API using the AWS CLI

To create a private API using the AWS CLI, call the `create-rest-api` command:

```
aws apigateway create-rest-api \
        --name 'Simple PetStore (AWS CLI, Private)' \
        --description 'Simple private PetStore API' \
        --region us-west-2 \
        --endpoint-configuration '{ "types": ["PRIVATE"] }'
```

A successful call returns output similar to the following:

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "Simple private PetStore API",
    "endpointConfiguration": {
        "types": "PRIVATE"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (AWS CLI, Private)"
}
```

From here on, you can follow the same instructions given in the section called "Set up an edge-optimized API using AWS CLI commands" to set up methods and integrations for this API.

When you are ready to test your API, be sure to create a resource policy and attach it to the API as described in ???.

Create a private API using the AWS SDK for JavaScript

To create a private API by using the AWS SDK for JavaScript:

```
apig.createRestApi({
 name: "Simple PetStore (node.js SDK, private)",
 endpointConfiguration: {
  types: ['PRIVATE']
 },
 description: "Demo private API created using the AWS SDK for node.js",
 version: "0.00.001"
}, function(err, data){
 if (!err) {
  console.log('Create API succeeded:\n', data);
  restApiId = data.id;
 } else {
  console.log('Create API failed:\n', err);
 }
});
```

A successful call returns output similar to the following:

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "Demo private API created using the AWS SDK for node.js",
    "endpointConfiguration": {
        "types": "PRIVATE"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (node.js SDK, private)"
}
```

After completing the preceding steps, you can follow the instructions in the section called "Set up an edge-optimized API using the AWS SDK for Node.js" to set up methods and integrations for this API.

When you are ready to test your API, be sure to create a resource policy and attach it to the API as described in ???.

Set up a resource policy for a private API

Before your private API can be accessed, you need to create a resource policy and attach it to the API. This grants access to the API from your VPCs and VPC endpoints or from VPCs and VPC endpoints in other AWS accounts that you explicitly grant access.

To do this, follow the instructions in the section called "Create and attach an API Gateway resource policy to an API". In step 5, choose the **Source VPC** example. Replace `{{vpceID}}` (including the curly braces) with your VPC endpoint ID, and then choose **Save** to save your resource policy.

You should also consider attaching an endpoint policy to the VPC endpoint to specify the access that's being granted. For more information, see the section called "Use VPC endpoint policies for private APIs".

Deploy a private API using the API Gateway console

To deploy your private API, do the following in the API Gateway console:

1. Choose your API.
2. Choose **Deploy API**.
3. For **Stage**, select **New stage**.
4. For **Stage name**, enter a stage name.
5. (Optional) For **Description**, enter a description.
6. Choose **Deploy**.

Associate or disassociate a VPC endpoint with a private REST API

When you associate a VPC endpoint with your private API, API Gateway generates a new Route 53 ALIAS DNS record. You can use this record to invoke your private APIs just as you do your edge-optimized or Regional APIs without overriding a `Host` header or passing an `x-apigw-api-id` header.

The generated base URL is in the following format:

```
https://{rest-api-id}-{vpce-id}.execute-api.{region}.amazonaws.com/{stage}
```

Associating or disassociating a VPC endpoint with a private REST API requires you to update the API's configuration. You can perform this change using the API Gateway console, the AWS CLI, or an AWS SDK for API Gateway. The update operation may take few minutes to complete due to DNS propagation. During this time, your API is available, but DNS propagation for the newly generated DNS URLs may still be in progress. You may try creating a new deployment for your API, if even after several minutes your new URLs are not resolving in DNS.

Use the API Gateway console to associate a VPC endpoint with a private REST API

To associate an additional VPC endpoint with a private API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your private API.
3. In the main navigation pane, choose **Resource policy**.
4. Edit your resource policy to allow calls from your additional VPC endpoint.
5. In the main navigation pane, choose **API settings**.
6. In the **API details** section, choose **Edit**.
7. For **VPC endpoint IDs**, select additional VPC endpoint IDs.
8. Choose **Save**.
9. Redeploy your API for the changes to take effect.

Use the AWS CLI to associate a VPC endpoint with a private REST API

To associate VPC endpoints at the time of API creation, use the following command:

```
aws apigateway create-rest-api \
    --name Petstore \
    --endpoint-configuration '{ "types": ["PRIVATE"], "vpcEndpointIds" :
 ["vpce-0212a4ababd5b8c3e", "vpce-0393a628149c867ee"] }' \
    --region us-west-2
```

The output will look like the following:

```
{
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "PRIVATE"
        ],
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-0393a628149c867ee"
        ]
    },
    "id": "u67n3ov968",
    "createdDate": 1565718256,
    "name": "Petstore"
}
```

To associate VPC endpoints to an already created private API, use the following CLI command:

```
aws apigateway update-rest-api \
    --rest-api-id u67n3ov968 \
    --patch-operations "op='add',path='/endpointConfiguration/
vpcEndpointIds',value='vpce-01d622316a7df47f9'" \
    --region us-west-2
```

The output will look like the following:

```
{
    "name": "Petstore",
    "apiKeySource": "1565718256",
    "tags": {},
    "createdDate": 1565718256,
    "endpointConfiguration": {
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-0393a628149c867ee",
            "vpce-01d622316a7df47f9"
        ],
        "types": [
            "PRIVATE"
        ]
    },
    "id": "u67n3ov968"
}
```

Use the API Gateway console to disassociate a VPC endpoint from a private REST API

To disassociate a VPC endpoint from a private REST API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your private API.
3. In the main navigation pane, choose **Resource policy**.
4. Edit your resource policy to remove mentions of the VPC endpoint you want to dissociate from your private API.
5. In the main navigation pane, choose **API settings**.
6. In the **API details** section, choose **Edit**.
7. For **VPC endpoint IDs**, choose the **X** to dissociate the VPC endpoint.
8. Choose **Save**.
9. Redeploy your API for the changes to take effect.

Use the AWS CLI to disassociate a VPC endpoint from a private REST API

To disassociate a VPC endpoint from a private API, use the following CLI command:

```
aws apigateway update-rest-api \
    --rest-api-id u67n3ov968 \
    --patch-operations "op='remove',path='/endpointConfiguration/
vpcEndpointIds',value='vpce-0393a628149c867ee'" \
    --region us-west-2
```

The output will look like the following:

```
{
    "name": "Petstore",
    "apiKeySource": "1565718256",
    "tags": {},
    "createdDate": 1565718256,
    "endpointConfiguration": {
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-01d622316a7df47f9"
        ],
        "types": [
            "PRIVATE"
        ]
    },
    "id": "u67n3ov968"
}
```

Monitoring REST APIs

In this section, you can learn how to monitor your API by using CloudWatch metrics, CloudWatch Logs, Firehose, and AWS X-Ray. By combining CloudWatch execution logs and CloudWatch metrics, you can log errors and execution traces, and monitor your API's performance. You might also want to log API calls to Firehose. You can also use AWS X-Ray to trace calls through the downstream services that make up your API.

Note

API Gateway might not generate logs and metrics in the following cases:

- 413 Request Entity Too Large errors
- Excessive 429 Too Many Requests errors
- 400 series errors from requests sent to a custom domain that has no API mapping
- 500 series errors caused by internal failures

API Gateway will not generate logs and metrics when testing a REST API method. The CloudWatch entries are simulated. For more information, see the section called “Use the console to test a REST API method”.

Topics

- Monitoring REST API execution with Amazon CloudWatch metrics
- Setting up CloudWatch logging for a REST API in API Gateway
- Logging API calls to Amazon Data Firehose

- Tracing user requests to REST APIs using X-Ray

Monitoring REST API execution with Amazon CloudWatch metrics

You can monitor API execution by using CloudWatch, which collects and processes raw data from API Gateway into readable, near-real-time metrics. These statistics are recorded for a period of 15 months so you can access historical information and gain a better perspective on how your web application or service is performing. By default, API Gateway metric data is automatically sent to CloudWatch in one-minute periods. For more information, see What Is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

The metrics reported by API Gateway provide information that you can analyze in different ways. The following list shows some common uses for the metrics that are suggestions to get you started:

- Monitor the **IntegrationLatency** metrics to measure the responsiveness of the backend.
- Monitor the **Latency** metrics to measure the overall responsiveness of your API calls.
- Monitor the **CacheHitCount** and **CacheMissCount** metrics to optimize cache capacities to achieve a desired performance.

Topics

Amazon API Gateway dimensions and metrics

The metrics and dimensions that API Gateway sends to Amazon CloudWatch are listed below. For more information, see Monitoring REST API execution with Amazon CloudWatch metrics.

API Gateway metrics

Amazon API Gateway sends metric data to CloudWatch every minute.

The `AWS/ApiGateway` namespace includes the following metrics.

Metric	Description
4XXError	The number of client-side errors captured in a given period. API Gateway counts modified gateway response status codes as 4XXError errors. The `Sum` statistic represents this metric, namely, the total count of the 4XXError errors in the given period. The `Average` statistic represents the 4XXError error rate, namely, the total count of the 4XXError errors divided by the total number of requests during the period. The denominator corresponds to the Count metric (below). Unit: Count
5XXError	The number of server-side errors captured in a given period. The `Sum` statistic represents this metric, namely, the total count of the 5XXError errors in the given period. The `Average` statistic represents the 5XXError error rate, namely, the total count of the 5XXError errors divided by the total number of requests during the period. The denominator corresponds to the Count metric (below). Unit: Count
CacheHitCount	The number of requests served from the API cache in a given period. The `Sum` statistic represents this metric, namely, the total count of the cache hits in the given period. The `Average` statistic represents the cache hit rate, namely, the total count of the cache hits divided by the total number of requests during the

Metric	Description
	period. The denominator corresponds to the Count metric (below). Unit: Count
CacheMissCount	The number of requests served from the backend in a given period, when API caching is enabled. The `Sum` statistic represents this metric, namely, the total count of the cache misses in the given period. The `Average` statistic represents the cache miss rate, namely, the total count of the cache misses divided by the total number of requests during the period. The denominator corresponds to the Count metric (below). Unit: Count
Count	The total number API requests in a given period. The `SampleCount` statistic represents this metric. Unit: Count
IntegrationLatency	The time between when API Gateway relays a request to the backend and when it receives a response from the backend. Unit: Millisecond
Latency	The time between when API Gateway receives a request from a client and when it returns a response to the client. The latency includes the integration latency and other API Gateway overhead. Unit: Millisecond

Dimensions for metrics

You can use the dimensions in the following table to filter API Gateway metrics.

> **Note**
>
> API Gateway removes non-ASCII characters from the ApiName dimension before sending metrics to CloudWatch. If the APIName contains no ASCII characters, the API ID is used as the ApiName.

Dimension	Description
ApiName	Filters API Gateway metrics for the REST API with the specified API name.
ApiName, Method, Resource, Stage	Filters API Gateway metrics for the API method with the specified API name, stage, resource, and method. API Gateway will not send these metrics unless you have explicitly enabled detailed CloudWatch metrics. In the console, choose a stage, and then for **Logs and tracing**, select **Edit**. Select **Detailed metrics**, and then choose **Save changes**. Alternatively, you can call the update-stage AWS CLI command to update the `metricsEnabled` property to `true`. Enabling these metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing.
ApiName, Stage	Filters API Gateway metrics for the API stage resource with the specified API name and stage.

View CloudWatch metrics with the API dashboard in API Gateway

You can use the API dashboard in the API Gateway Console to display the CloudWatch metrics of your deployed API in API Gateway. These are shown as a summary of API activity over time.

Topics

- Prerequisites
- Examine API activities in the dashboard

Prerequisites

1. You must have an API created in API Gateway. Follow the instructions in Creating a REST API in Amazon API Gateway.
2. You must have the API deployed at least once. Follow the instructions in Deploying a REST API in Amazon API Gateway.

Examine API activities in the dashboard

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an API.
3. In the main navigation pane, choose **Dashboard**.
4. For **Stage**, choose the desired stage.
5. Choose **Date range** to specify a range of dates.
6. Refresh, if needed, and view individual metrics displayed in separate graphs titled **API calls**, **Latency**, **Integration latency**, **Latency**, **4xx error** and **5xx error**.

> **Tip**
>
> To examine method-level CloudWatch metrics, make sure that you have enabled CloudWatch Logs on a method level. For more information about how to set up method-level logging, see Update stage settings using the API Gateway console.

View API Gateway metrics in the CloudWatch console

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

To view API Gateway metrics using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. If necessary, change the AWS Region. From the navigation bar, select the Region where your AWS resources reside. For more information, see Regions and Endpoints.
3. In the navigation pane, choose **Metrics**.
4. In the **All metrics** tab, choose **API Gateway**.
5. To view metrics by stage, choose the **By Stage** panel. And then select desired APIs and metric names.
6. To view metrics by specific API, choose the **By Api Name** panel. And then select desired APIs and metric names.

To view metrics using the AWS CLI

1. At a command prompt, use the following command to list metrics:

   ```
   aws cloudwatch list-metrics --namespace "AWS/ApiGateway"
   ```

2. To view a specific statistics (for example, `Average`) over a period of time of a 5 minutes intervals, call the following command:

   ```
   aws cloudwatch get-metric-statistics --namespace AWS/ApiGateway --metric-name Count
    --start-time 2011-10-03T23:00:00Z --end-time 2017-10-05T23:00:00Z --period 300 --
   statistics Average
   ```

View API Gateway log events in the CloudWatch console

Prerequisites

1. You must have an API created in API Gateway. Follow the instructions in Creating a REST API in Amazon API Gateway.

2. You must have the API deployed and invoked at least once. Follow the instructions in Deploying a REST API in Amazon API Gateway and Invoking a REST API in Amazon API Gateway.
3. You must have CloudWatch Logs enabled for a stage. Follow the instructions in Setting up CloudWatch logging for a REST API in API Gateway.

To view logged API requests and responses using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. If necessary, change the AWS Region. From the navigation bar, select the Region where your AWS resources reside. For more information, see Regions and Endpoints.
3. In the navigation pane, choose **Logs**, **Log groups**.
4. Under the **Log Groups** table, choose a log group of the **API-Gateway-Execution-Logs_{rest-api-id}/{stage-name}** name.
5. Under the **Log Streams** table, choose a log stream. You can use the timestamp to help locate the log stream of your interest.
6. Choose **Text** to view raw text or choose **Row** to view the event row by row.

⚠ **Important**

CloudWatch lets you delete log groups or streams. Do not manually delete API Gateway API log groups or streams; let API Gateway manage these resources. Manually deleting log groups or streams may cause API requests and responses not to be logged. If that happens, you can delete the entire log group for the API and redeploy the API. This is because API Gateway creates log groups or log streams for an API stage at the time when it is deployed.

Monitoring tools in AWS

AWS provides various tools that you can use to monitor API Gateway. You can configure some of these tools to do the monitoring for you automatically, while other tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated monitoring tools in AWS

You can use the following automated monitoring tools to watch API Gateway and report when something is wrong:

- **Amazon CloudWatch Alarms –** Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see Monitoring REST API execution with Amazon CloudWatch metrics.
- **Amazon CloudWatch Logs –** Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see Monitoring Log Files in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events –** Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see What is Amazon CloudWatch Events in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring –** Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Working with CloudTrail Log Files in the *AWS CloudTrail User Guide*.

Manual monitoring tools

Another important part of monitoring API Gateway involves manually monitoring those items that the CloudWatch alarms don't cover. The API Gateway, CloudWatch, and other AWS console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on API execution.

- API Gateway dashboard shows the following statistics for a given API stage during a specified period of time:
  - **API Calls**
  - **Cache Hit**, only when API caching is enabled.
  - **Cache Miss**, only when API caching is enabled.
  - **Latency**

  - **Integration Latency**
  - **4XX Error**
  - **5XX Error**
- The CloudWatch home page shows:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, you can use CloudWatch to do the following:

  - Create customized dashboards to monitor the services you care about
  - Graph metric data to troubleshoot issues and discover trends
  - Search and browse all your AWS resource metrics
  - Create and edit alarms to be notified of problems

Creating CloudWatch alarms to monitor API Gateway

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period you specify, and performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

Setting up CloudWatch logging for a REST API in API Gateway

To help debug issues related to request execution or client access to your API, you can enable Amazon CloudWatch Logs to log API calls. For more information about CloudWatch, see the section called “CloudWatch metrics”.

CloudWatch log formats for API Gateway

There are two types of API logging in CloudWatch: execution logging and access logging. In execution logging, API Gateway manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams any caller's requests and responses.

The logged data includes errors or execution traces (such as request or response parameter values or payloads), data used by Lambda authorizers (formerly known as custom authorizers), whether API keys are required, whether usage plans are enabled, and so on.

When you deploy an API, API Gateway creates a log group and log streams under the log group. The log group is named following the `API-Gateway-Execution-Logs_{rest-api-id}/{stage_name}` format. Within each log group, the logs are further divided into log streams, which are ordered by **Last Event Time** as logged data is reported.

In access logging, you, as an API developer, want to log who has accessed your API and how the caller accessed the API. You can create your own log group or choose an existing log group that could be managed by API Gateway. To specify the access details, you select `$context` variables, a log format, and a log group destination.

The access log format must include at least `$context.requestId` or `$context.extendedRequestId`. As a best practice, include `$context.requestId` and `$context.extendedRequestId` in your log format.

`$context.requestId`

This logs the value in the `x-amzn-RequestId` header. Clients can override the value in the `x-amzn-RequestId` header with a value in the format of a universally unique identifier (UUID). API Gateway returns this request ID in the `x-amzn-RequestId` response header. API Gateway replaces overridden request IDs that aren't in the format of a UUID with `UUID_REPLACED_INVALID_REQUEST_ID` in your access logs.

`$context.extendedRequestId`

The extendedRequestID is a unique ID that API Gateway generates. API Gateway returns this request ID in the `x-amz-apigw-id` response header. An API caller can't provide or override this request ID. You might need to provide this value to AWS Support to help troubleshoot your API. For more information, see the section called "`$context` Variables for data models, authorizers, mapping templates, and CloudWatch access logging".

> **Note**
>
> Only `$context` variables are supported.

Choose a log format that is also adopted by your analytic backend, such as Common Log Format (CLF), JSON, XML, or CSV. You can then feed the access logs to it directly to have your metrics computed and rendered. To define the log format, set the log group ARN on the accessLogSettings/destinationArn property on the stage. You can obtain a log group ARN in the CloudWatch console. To define the access log format, set a chosen format on the accessLogSetting/format property on the stage.

Examples of some commonly used access log formats are shown in the API Gateway console and are listed as follows.

- CLF (Common Log Format):

```
$context.identity.sourceIp $context.identity.caller $context.identity.user
 [$context.requestTime]"$context.httpMethod $context.resourcePath
 $context.protocol" $context.status $context.responseLength $context.requestId
 $context.extendedRequestId
```

- JSON:

```
{ "requestId":"$context.requestId",
 "extendedRequestId":"$context.extendedRequestId","ip": "$context.identity.sourceIp",
 "caller":"$context.identity.caller", "user":"$context.identity.user",
 "requestTime":"$context.requestTime", "httpMethod":"$context.httpMethod",
 "resourcePath":"$context.resourcePath", "status":"$context.status",
 "protocol":"$context.protocol", "responseLength":"$context.responseLength" }
```

- XML:

```
<request id="$context.requestId"> <extendedRequestId>$context.extendedRequestId</
extendedRequestId> <ip>$context.identity.sourceIp</ip> <caller>
$context.identity.caller</caller> <user>$context.identity.user</user> <requestTime>
$context.requestTime</requestTime> <httpMethod>$context.httpMethod</httpMethod>
 <resourcePath>$context.resourcePath</resourcePath> <status>$context.status</status>
 <protocol>$context.protocol</protocol> <responseLength>$context.responseLength</
responseLength> </request>
```

- CSV (comma-separated values):

```
$context.identity.sourceIp,$context.identity.caller,$context.identity.user,
$context.requestTime,$context.httpMethod,$context.resourcePath,$context.protocol,
$context.status,$context.responseLength,$context.requestId,$context.extendedRequestId
```

Permissions for CloudWatch logging

To enable CloudWatch Logs, you must grant API Gateway permission to read and write logs to CloudWatch for your account. The `AmazonAPIGatewayPushToCloudWatchLogs` managed policy (with an ARN of `arn:aws:iam::aws:policy/service-role/AmazonAPIGatewayPushToCloudWatchLogs`) has all the required permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:GetLogEvents",
                "logs:FilterLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

> **Note**
>
> API Gateway calls AWS Security Token Service in order to assume the IAM role, so make sure that AWS STS is enabled for the Region. For more information, see Managing AWS STS in an AWS Region.

To grant these permissions to your account, create an IAM role with `apigateway.amazonaws.com` as its trusted entity, attach the preceding policy to the IAM role, and set the IAM role ARN on the cloudWatchRoleArn property on your Account. You must set the cloudWatchRoleArn property separately for each AWS Region in which you want to enable CloudWatch Logs.

If you receive an error when setting the IAM role ARN, check your AWS Security Token Service account settings to make sure that AWS STS is enabled in the Region that you're using. For more information about enabling AWS STS, see Managing AWS STS in an AWS Region in the *IAM User Guide*.

Set up CloudWatch API logging using the API Gateway console

To set up CloudWatch API logging, you must have deployed the API to a stage. You must also have configured an appropriate CloudWatch Logs role ARN for your account.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. On the main navigation pane, choose **Settings**, and then under **Logging**, choose **Edit**.
3. For **CloudWatch log role ARN**, enter an ARN of an IAM role with appropriate permissions. You need to do this once for each AWS account that creates APIs using API Gateway.
4. In the main navigation pane, choose **APIs**, and then do one of the following:

    a. Choose an existing API, and then choose a stage.

    b. Create an API, and then deploy it to a stage.
5. In the main navigation pane, choose **Stages**.
6. In the **Logs and tracing** section, choose **Edit**.
7. To enable execution logging:

    a. Select a logging level from the **CloudWatch Logs** dropdown menu. The logging levels are the following:

    - **Off** – Logging is not turned on for this stage.
    - **Errors only** – Logging is enabled for errors only.
    - **Errors and info logs** – Logging is enabled for all events.
    - **Full request and response logs** – Detailed logging is enabled for all events. This can be useful to troubleshoot APIs, but can result in logging sensitive data.

    > **Note**
    >
    > We recommend that you don't use **Full request and response logs** for production APIs.

    b. If desired, select **Detailed metrics** to turn on detailed CloudWatch metrics.

For more information about CloudWatch metrics, see the section called "CloudWatch metrics".

8. To enable access logging:

   a. Turn on **Custom access logging**.

   b. For **Access log destination ARN**, enter the ARN of a log group. The ARN format is `arn:aws:logs:{region}:{account-id}:log-group:log-group-name`.

   c. For **Log Format**, enter a log format. You can choose **CLF**, **JSON**, **XML**, or **CSV**. To learn more about example log formats, see the section called "CloudWatch log formats for API Gateway".

9. Choose **Save changes**.

> **Note**
>
> You can enable execution logging and access logging independently of each other.

API Gateway is now ready to log requests to your API. You don't need to redeploy the API when you update the stage settings, logs, or stage variables.

Set up CloudWatch API logging using AWS CloudFormation

Use the following example AWS CloudFormation template to create an Amazon CloudWatch Logs log group and configure execution and access logging for a stage. To enable CloudWatch Logs, you must grant API Gateway permission to read and write logs to CloudWatch for your account. To learn more, see Associate account with IAM role in the *AWS CloudFormation User Guide*.

```
TestStage:
  Type: AWS::ApiGateway::Stage
  Properties:
    StageName: test
    RestApiId: !Ref MyAPI
    DeploymentId: !Ref Deployment
    Description: "test stage description"
    MethodSettings:
      - ResourcePath: "/*"
        HttpMethod: "*"
        LoggingLevel: INFO
      AccessLogSetting:
        DestinationArn: !GetAtt MyLogGroup.Arn
        Format: $context.extendedRequestId $context.identity.sourceIp
$context.identity.caller $context.identity.user [$context.requestTime]
"$context.httpMethod $context.resourcePath $context.protocol" $context.status
$context.responseLength $context.requestId
 MyLogGroup:
   Type: AWS::Logs::LogGroup
   Properties:
     LogGroupName: !Join
       - '-'
       - - !Ref MyAPI
         - access-logs
```

Logging API calls to Amazon Data Firehose

To help debug issues related to client access to your API, you can log API calls to Amazon Data Firehose. For more information about Firehose, see What Is Amazon Data Firehose?.

For access logging, you can only enable CloudWatch or Firehose—you can't enable both. However, you can enable CloudWatch for execution logging and Firehose for access logging.

Topics

- Firehose log formats for API Gateway
- Permissions for Firehose logging
- Set up Firehose access logging by using the API Gateway console

Firehose log formats for API Gateway

Firehose logging uses the same format as CloudWatch logging.

Permissions for Firehose logging

When Firehose access logging is enabled on a stage, API Gateway creates a service-linked role in your account if the role doesn't exist already. The role is named `AWSServiceRoleForAPIGateway` and has the `APIGatewayServiceRolePolicy` managed policy attached to it. For more information about service-linked roles, see Using Service-Linked Roles.

> **Note**
>
> The name of your Firehose stream must be `amazon-apigateway-{your-stream-name}`.

Set up Firehose access logging by using the API Gateway console

To set up API logging, you must have deployed the API to a stage. You must also have created a Firehose stream.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
    a. Choose an existing API, and then choose a stage.
    b. Create an API and deploy it to a stage.
3. In the main navigation pane, choose **Stages**.
4. In the **Logs and tracing** section, choose **Edit**.
5. To enable access logging to a Firehose stream:
    a. Turn on **Custom access logging**.
    b. For **Access log destination ARN**, enter the ARN of a Firehose stream. The ARN format is `arn:aws:firehose:{region}:{account-id}:deliverystream/amazon-apigateway-{your-stream-name}`.

    > **Note**
    >
    > The name of your Firehose stream must be `amazon-apigateway-{your-stream-name}`.

    c. For **Log format**, enter a log format. You can choose **CLF**, **JSON**, **XML**, or **CSV**. To learn more about example log formats, see the section called "CloudWatch log formats for API Gateway".
6. Choose **Save changes**.

API Gateway is now ready to log requests to your API to Firehose. You don't need to redeploy the API when you update the stage settings, logs, or stage variables.

Tracing user requests to REST APIs using X-Ray

You can use AWS X-Ray to trace and analyze user requests as they travel through your Amazon API Gateway REST APIs to the underlying services. API Gateway supports X-Ray tracing for all API Gateway REST API endpoint types: Regional, edge-optimized, and private. You can use X-Ray with Amazon API Gateway in all AWS Regions where X-Ray is available.

Because X-Ray gives you an end-to-end view of an entire request, you can analyze latencies in your APIs and their backend services. You can use an X-Ray service map to view the latency of an entire request and that of the downstream services that are integrated with X-Ray. You can also configure sampling rules to tell X-Ray which requests to record and at what sampling rates, according to criteria that you specify.

If you call an API Gateway API from a service that's already being traced, API Gateway passes the trace through, even if X-Ray tracing isn't enabled on the API.

You can enable X-Ray for an API stage by using the API Gateway console, or by using the API Gateway API or CLI.

Topics

Setting up AWS X-Ray with API Gateway REST APIs

In this section you can find detailed information on how to set up AWS X-Ray with API Gateway REST APIs.

Topics

X-Ray tracing modes for API Gateway

The path of a request through your application is tracked with a trace ID. A trace collects all of the segments generated by a single request, typically an HTTP `GET` or `POST` request.

There are two modes of tracing for an API Gateway API:

- **Passive**: This is the default setting if you have not enabled X-Ray tracing on an API stage. This approach means that the API Gateway API is only traced if X-Ray has been enabled on an upstream service.
- **Active**: When an API Gateway API stage has this setting, API Gateway automatically samples API invocation requests, based on the sampling algorithm specified by X-Ray.

  When active tracing is enabled on a stage, API Gateway creates a service-linked role in your account, if the role does not exist already. The role is named `AWSServiceRoleForAPIGateway` and will have the `APIGatewayServiceRolePolicy` managed policy attached to it. For more information about service-linked roles, see Using Service-Linked Roles.

> **Note**
>
> X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your API receives. The default sampling algorithm is 1 request per second, with 5 percent of requests sampled past that limit.

You can change the tracing mode for your API by using the API Gateway management console, the API Gateway CLI, or an AWS SDK.

Permissions for X-Ray tracing

When you enable X-Ray tracing on a stage, API Gateway creates a service-linked role in your account, if the role does not exist already. The role is named `AWSServiceRoleForAPIGateway` and will have the `APIGatewayServiceRolePolicy` managed policy attached to it. For more information about service-linked roles, see Using Service-Linked Roles.

Enabling X-Ray tracing in the API Gateway console

You can use the Amazon API Gateway console to enable active tracing on an API stage.

These steps assume that you have already deployed the API to a stage.

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API, and then in the main navigation pane, choose **Stages**.
3. In the **Stages** pane, choose a stage.
4. In the **Logs and tracing** section, choose **Edit**.
5. To enable active X-Ray tracing, select **X-Ray tracing** to turn on X-Ray tracing.
6. Choose **Save changes**.

Once you've enabled X-Ray for your API stage, you can use the X-Ray management console to view the traces and service maps.

Enabling AWS X-Ray tracing using the API Gateway CLI

To use the AWS CLI to enable active X-Ray tracing for an API stage when you create the stage, call the `create-stage` command as in the following example:

```
aws apigateway create-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --deployment-id {deployment-id} \
    --region {region} \
    --tracing-enabled=true
```

Following is example output for a successful invocation:

```
{
    "tracingEnabled": true,
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533849811,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

To use the AWS CLI to disable active X-Ray tracing for an API stage when you create the stage, call the `create-stage` command as in the following example:

```
aws apigateway create-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --deployment-id {deployment-id} \
    --region {region} \
    --tracing-enabled=false
```

Following is example output for a successful invocation:

```
{
    "tracingEnabled": false,
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533849811,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

To use the AWS CLI to enable active X-Ray tracing for an API that's already been deployed, call the `update-stage` command as follows:

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --patch-operations op=replace,path=/tracingEnabled,value=true
```

To use the AWS CLI to disable active X-Ray tracing for an API that's already been deployed, call the `update-stage` command as in the following example:

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --region {region} \
    --patch-operations op=replace,path=/tracingEnabled,value=false
```

Following is example output for a successful invocation:

```
{
    "tracingEnabled": false,
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533850033,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

Once you've enabled X-Ray for your API stage, use the X-Ray CLI to retrieve trace information. For more information, see Using the AWS X-Ray API with the AWS CLI.

Using AWS X-Ray service maps and trace views with API Gateway

In this section you can find detailed information on how to use AWS X-Ray service maps and trace views with API Gateway.

For detailed information about service maps and trace views, and how to interpret them, see AWS X-Ray Console.

Topics

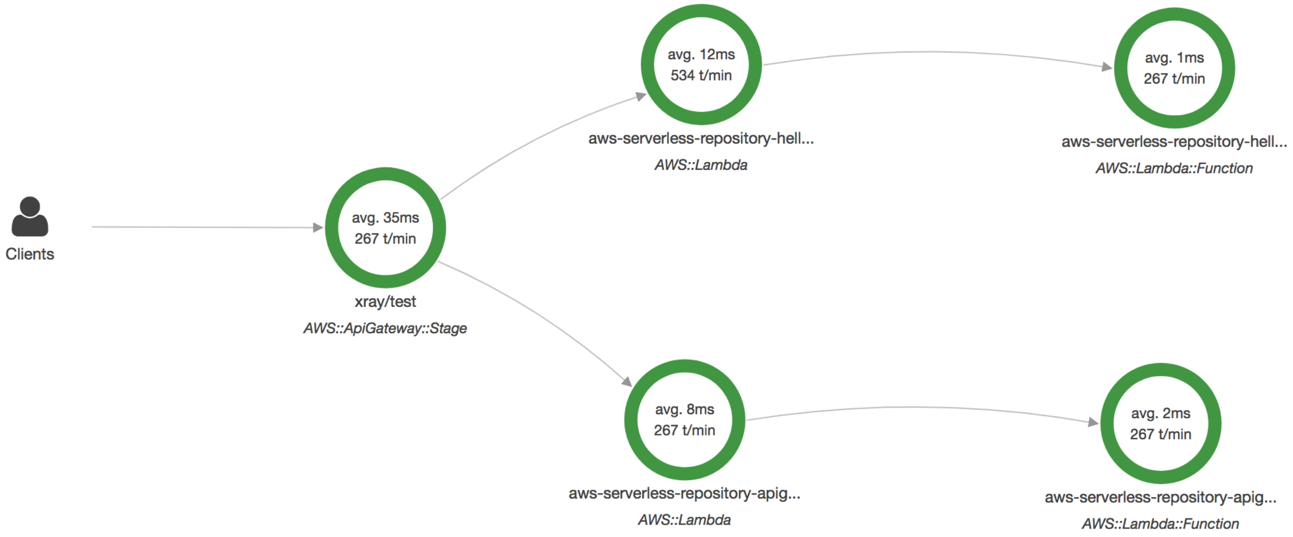
- Example X-Ray service map
- Example X-Ray trace view

Example X-Ray service map

AWS X-Ray service maps show information about your API and all of its downstream services. When X-Ray is enabled for an API stage in API Gateway, you'll see a node in the service map containing information about the overall time spent in the API Gateway service. You can get detailed information about the response status and a histogram of the API response time for the selected timeframe. For APIs integrating with AWS services such as AWS Lambda and Amazon DynamoDB, you will see more nodes providing performance metrics related to those services. There will be a service map for each API stage.

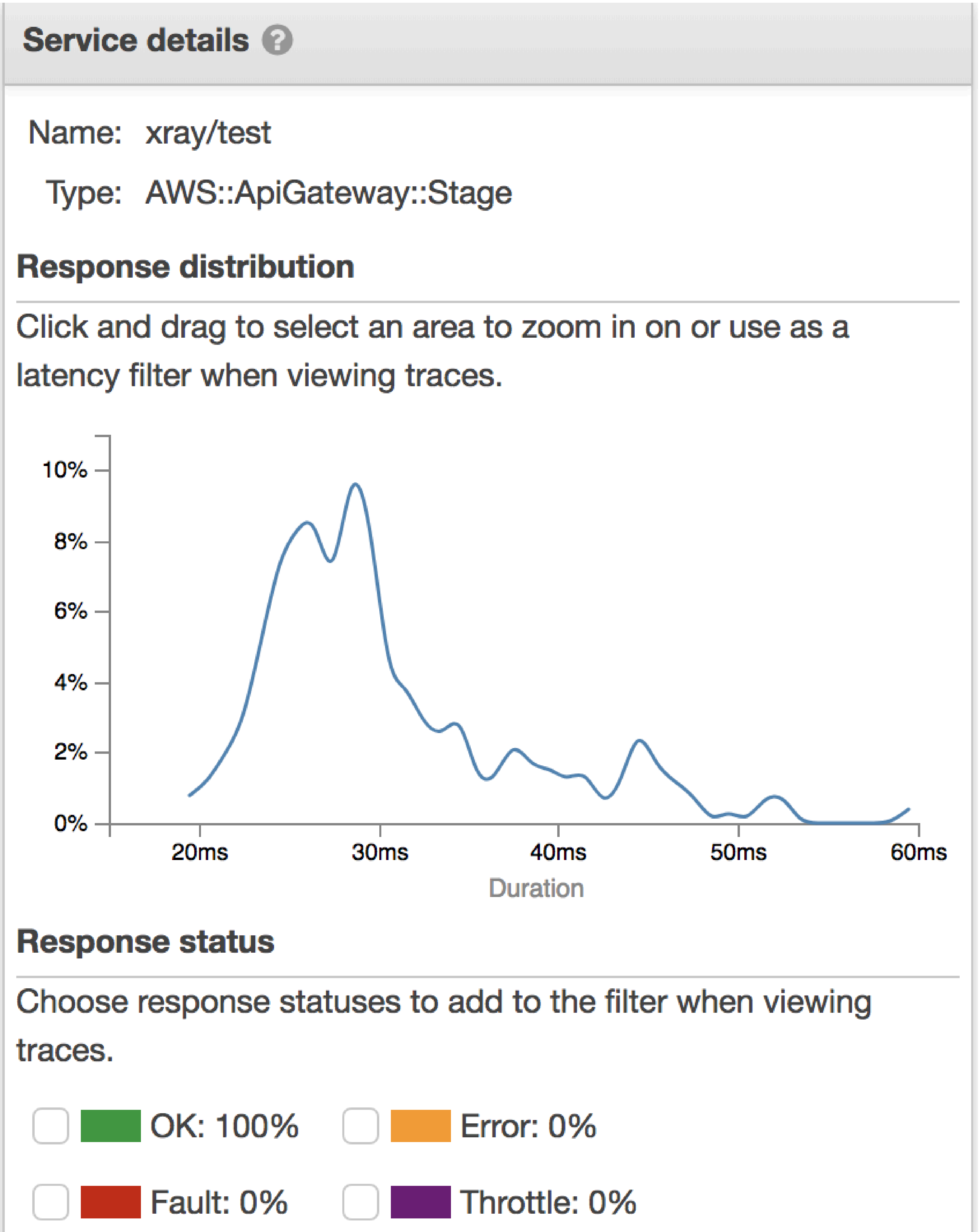
The following example shows a service map for the `test` stage of an API called `xray`. This API has a Lambda integration with a Lambda authorizer function and a Lambda backend function. The nodes represent the API Gateway service, the Lambda service, and the two Lambda functions.

For a detailed explanation of service map structure, see Viewing the Service Map.



From the service map, you can zoom in to see a trace view of your API stage. The trace will display in-depth information regarding your API, represented as segments and subsegments. For example, the trace for the service map shown above would include segments for the Lambda service and Lambda function. For more information, see Lambda as an AWS X-Ray Trace.

If you choose a node or edge on an X-Ray service map, the X-Ray console shows a latency distribution histogram. You can use a latency histogram to see how long it takes for a service to complete its requests. Following is a histogram of the API Gateway stage named `xray/test` in the previous service map. For a detailed explanation of latency distribution histograms, see Using Latency Histograms in the AWS X-Ray Console.



Service details

Name: xray/test

Type: AWS::ApiGateway::Stage

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

Response status

Choose response statuses to add to the filter when viewing traces.

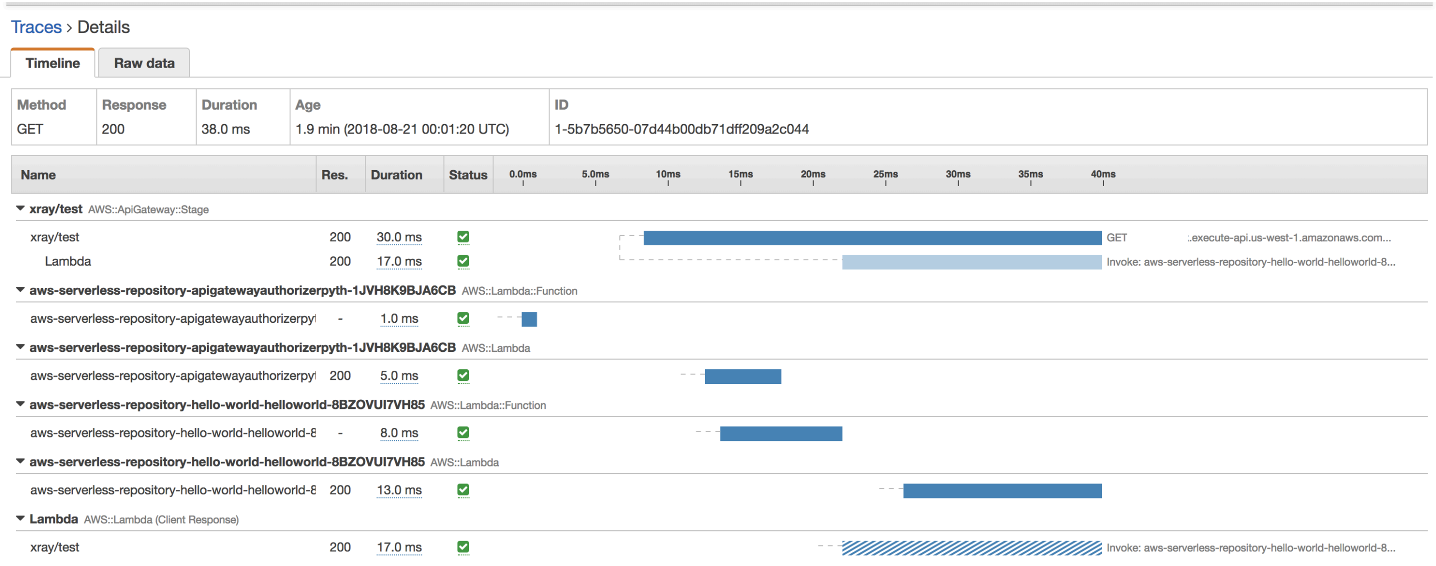
□ 🟩 OK: 100% □ 🟧 Error: 0%

□ 🟥 Fault: 0% □ 🟪 Throttle: 0%

Example X-Ray trace view

The following diagram shows a trace view generated for the example API described above, with a Lambda backend function and a Lambda authorizer function. A successful API method request is shown with a response code of 200.

For a detailed explanation of trace views, see Viewing Traces.



Traces › Details

Timeline | Raw data

Method	Response	Duration	Age	ID
GET	200	38.0 ms	1.9 min (2018-08-21 00:01:20 UTC)	1-5b7b5650-07d44b00db71dff209a2c044

Name	Res.	Duration	Status
xray/test AWS::ApiGateway::Stage			
xray/test	200	30.0 ms	✓
Lambda	200	17.0 ms	✓
aws-serverless-repository-apigatewayauthorizerpyth-1JVH8K9BJA6CB AWS::Lambda::Function			
aws-serverless-repository-apigatewayauthorizerpyt	-	1.0 ms	✓
aws-serverless-repository-apigatewayauthorizerpyth-1JVH8K9BJA6CB AWS::Lambda			
aws-serverless-repository-apigatewayauthorizerpyt	200	5.0 ms	✓
aws-serverless-repository-hello-world-helloworld-8BZOVUI7VH85 AWS::Lambda::Function			
aws-serverless-repository-hello-world-helloworld-8	-	8.0 ms	✓
aws-serverless-repository-hello-world-helloworld-8BZOVUI7VH85 AWS::Lambda			
aws-serverless-repository-hello-world-helloworld-8	200	13.0 ms	✓
Lambda AWS::Lambda (Client Response)			
xray/test	200	17.0 ms	✓

0.0ms
5.0ms
10ms
15ms
20ms
25ms
30ms
35ms
40ms
GET .execute-api.us-west-1.amazonaws.com...
Invoke: aws-serverless-repository-hello-world-helloworld-8...
Invoke: aws-serverless-repository-hello-world-helloworld-8...

Configuring AWS X-Ray sampling rules for API Gateway APIs

You can use AWS X-Ray console or SDK to configure sampling rules for your Amazon API Gateway API. A sampling rule specifies which requests X-Ray should record for your API. By customizing sampling rules, you can control the amount of data that you record, and modify sampling behavior on the fly without modifying or redeploying your code.

Before you specify your X-Ray sampling rules, read the following topics in the X-Ray Developer Guide:

- Configuring Sampling Rules in the AWS X-Ray Console
- Using Sampling Rules with the X-Ray API

Topics

- X-Ray sampling rule option values for API Gateway APIs
- X-Ray sampling rule examples

X-Ray sampling rule option values for API Gateway APIs

The following X-Ray sampling options are relevant for API Gateway. String values can use wildcards to match a single character (?) or zero or more characters (*). For more details, including a detailed explanation of how the **Reservoir** and **Rate** settings are used, Configuring Sampling Rules in the AWS X-Ray Console.

- **Rule name** (string) — A unique name for the rule.
- **Priority** (integer between 1 and 9999) — The priority of the sampling rule. Services evaluate rules in ascending order of priority, and make a sampling decision with the first rule that matches.
- **Reservoir** (nonnegative integer) — A fixed number of matching requests to instrument per second, before applying the fixed rate. The reservoir is not used directly by services, but applies to all services using the rule collectively.
- **Rate** (number between 0 and 100) — The percentage of matching requests to instrument, after the reservoir is exhausted.
- **Service name** (string) — API stage name, in the form ***{api-name}*/*{stage-name}***. For example, if you were to deploy the PetStore sample API to a stage named `test`, the **Service name** value to specify in your sampling rule would be **`pets/test`**.
- **Service type** (string) — For an API Gateway API, either **`AWS::ApiGateway::Stage`** or **`AWS::ApiGateway::*`** can be specified.
- **Host** (string) — The hostname from the HTTP host header. Set this to **`*`** to match against all hostnames. Or you can specify a full or partial hostname to match, for example, **`api.example.com`** or **`*.example.com`**.
- **Resource ARN** (string) — The ARN of the API stage, for example, **`arn:aws:apigateway:`*`region`*`::/restapis/`*`api-id`*`/stages/`*`stage-name`***.

  The stage name can be obtained from the console or the API Gateway CLI or API. For more information about ARN formats, see the Amazon Web Services General Reference.
- **HTTP method** (string) — The method to be sampled, for example, **`GET`**.
- **URL path** (string) — The URL path of the request.
- (optional) **Attributes** (key and value) — Headers from the original HTTP request, for example, **`Connection`**, **`Content-Length`**, or **`Content-Type`**. Each attribute value can be up to 32 characters long.

X-Ray sampling rule examples

Sampling rule example #1

This rule samples all `GET` requests for the `testxray` API at the `test` stage.

- **Rule name — `test-sampling`**
- **Priority — `17`**
- **Reservoir size — `10`**
- **Fixed rate — `10`**
- **Service name — `testxray/test`**
- **Service type — `AWS::ApiGateway::Stage`**
- **HTTP method — `GET`**
- **Resource ARN — `*`**
- **Host — `*`**

Sampling rule example #2

This rule samples all requests for the `testxray` API at the `prod` stage.

- **Rule name — `prod-sampling`**
- **Priority — `478`**
- **Reservoir size — `1`**
- **Fixed rate — `60`**
- **Service name — `testxray/prod`**
- **Service type — `AWS::ApiGateway::Stage`**
- **HTTP method — `*`**
- **Resource ARN — `*`**
- **Host — `*`**
- **Attributes — `{}`**

Understanding AWS X-Ray traces for Amazon API Gateway APIs

This section discusses AWS X-Ray trace segments, subsegments, and other trace fields for Amazon API Gateway APIs.

Before you read this section, review the following topics in the X-Ray Developer Guide:

- AWS X-Ray Console
- AWS X-Ray Segment Documents
- X-Ray Concepts

Topics

Examples of trace objects for an API Gateway API

This section discusses some of the objects you may see in a trace for an API Gateway API.

Annotations

Annotations can appear in segments and subsegments. They are used as filtering expressions in sampling rules to filter traces. For more information, see Configuring Sampling Rules in the AWS X-Ray Console.

Following is an example of an `annotations` object, in which an API stage is identified by the API ID and the API stage name:

```
"annotations": {
    "aws:api_id": "a1b2c3d4e5",
    "aws:api_stage": "dev"
}
```

AWS resource data

The `aws` object appears only in segments. Following is an example of an `aws` object that matches the Default sampling rule. For an in-depth explanation of sampling rules, see Configuring Sampling Rules in the AWS X-Ray Console.

```
"aws": {
    "xray": {
        "sampling_rule_name": "Default"
    },
    "api_gateway": {
        "account_id": "123412341234",
        "rest_api_id": "a1b2c3d4e5",
        "stage": "dev",
        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
    }
}
```

Understanding the trace

Following is a trace segment for an API Gateway stage. For a detailed explanation of the fields that make up the trace segment, see AWS X-Ray Segment Documents in the AWS X-Ray Developer Guide.

```
        {
            "Document": {
                "id": "a1b2c3d4a1b2c3d4",
                "name": "testxray/dev",
                "start_time": 1533928226.229,
                "end_time": 1533928226.614,
                "metadata": {
                    "default": {
                        "extended_request_id": "abcde12345abcde=",
                        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
                    }
                },
                "http": {
                    "request": {
                        "url": "https://example.com/dev?
username=demo&message=hellofromdemo/",
                        "method": "GET",
                        "client_ip": "192.0.2.0",
                        "x_forwarded_for": true
                    },
                    "response": {
                        "status": 200,
                        "content_length": 0
                    }
                },
                "aws": {
                    "xray": {
                        "sampling_rule_name": "Default"
                    },
                    "api_gateway": {
                        "account_id": "123412341234",
                        "rest_api_id": "a1b2c3d4e5",
                        "stage": "dev",
                        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
                    }
                },
                "annotations": {
                    "aws:api_id": "a1b2c3d4e5",
                    "aws:api_stage": "dev"
                },
                "trace_id": "1-a1b2c3d4-a1b2c3d4a1b2c3d4a1b2c3d4",
                "origin": "AWS::ApiGateway::Stage",
                "resource_arn": "arn:aws:apigateway:us-east-1::/restapis/a1b2c3d4e5/
stages/dev",
                "subsegments": [
                    {
                        "id": "abcdefgh12345678",
                        "name": "Lambda",
                        "start_time": 1533928226.233,
                        "end_time": 1533928226.6130002,
                        "http": {
                            "request": {
                                "url": "https://example.com/2015-03-31/functions/
arn:aws:lambda:us-east-1:123412341234:function:xray123/invocations",
                                "method": "GET"
                            },
                            "response": {
                                "status": 200,
                                "content_length": 62
                            }
                        },
                        "aws": {
                            "function_name": "xray123",
                            "region": "us-east-1",
                            "operation": "Invoke",
                            "resource_names": [
                                "xray123"
                            ]
                        },
                        "namespace": "aws"
                    }
                ]
            },
            "Id": "a1b2c3d4a1b2c3d4"
```

```
        }
```

Working with HTTP APIs

REST APIs and HTTP APIs are both RESTful API products. REST APIs support more features than HTTP APIs, while HTTP APIs are designed with minimal features so that they can be offered at a lower price. For more information, see the section called “Choosing between REST APIs and HTTP APIs ”.

You can use HTTP APIs to send requests to AWS Lambda functions or to any routable HTTP endpoint. For example, you can create an HTTP API that integrates with a Lambda function on the backend. When a client calls your API, API Gateway sends the request to the Lambda function and returns the function's response to the client.

HTTP APIs support OpenID Connect and OAuth 2.0 authorization. They come with built-in support for cross-origin resource sharing (CORS) and automatic deployments.

You can create HTTP APIs by using the AWS Management Console, the AWS CLI, APIs, AWS CloudFormation, or SDKs.

Topics

Developing an HTTP API in API Gateway

This section provides details about API Gateway capabilities that you need while you're developing your API Gateway APIs.

As you're developing your API Gateway API, you decide on a number of characteristics of your API. These characteristics depend on the use case of your API. For example, you might want to only allow certain clients to call your API, or you might want it to be available to everyone. You might want an API call to execute a Lambda function, make a database query, or call an application.

Topics

- Working with routes for HTTP APIs
- Controlling and managing access to an HTTP API in API Gateway
- Configuring integrations for HTTP APIs
- Configuring CORS for an HTTP API
- Transforming API requests and responses
- Working with OpenAPI definitions for HTTP APIs

Creating an HTTP API

To create a functional API, you must have at least one route, integration, stage, and deployment.

The following examples show how to create an API with an AWS Lambda or HTTP integration, a route, and a default stage that is configured to automatically deploy changes.

This guide assumes that you're already familiar with API Gateway and Lambda. For a more detailed guide, see *Getting started*.

Topics

- Create an HTTP API by using the AWS Management Console
- Create an HTTP API by using the AWS CLI

Create an HTTP API by using the AWS Management Console

1. Open the API Gateway console.
2. Choose **Create API**.
3. Under **HTTP API**, choose **Build**.
4. Choose **Add integration**, and then choose an AWS Lambda function or enter an HTTP endpoint.
5. For **Name**, enter a name for your API.
6. Choose **Review and create**.
7. Choose **Create**.

Now your API is ready to invoke. You can test your API by entering its invoke URL in a browser, or by using Curl.

```
curl https://api-id.execute-api.us-east-2.amazonaws.com
```

Create an HTTP API by using the AWS CLI

You can use quick create to create an API with a Lambda or HTTP integration, a default catch-all route, and a default stage that is configured to automatically deploy changes. The following command uses quick create to create an API that integrates with a Lambda function on the backend.

> **Note**
>
> To invoke a Lambda integration, API Gateway must have the required permissions. You can use a resource-based policy or an IAM role to grant API Gateway permissions to invoke a Lambda function. To learn more, see AWS Lambda Permissions in the *AWS Lambda Developer Guide*.

Example

```
aws apigatewayv2 create-api --name my-api --protocol-type HTTP --target
 arn:aws:lambda:us-east-2:123456789012:function:function-name
```

Now your API is ready to invoke. You can test your API by entering its invoke URL in a browser, or by using Curl.

```
curl https://api-id.execute-api.us-east-2.amazonaws.com
```

Working with routes for HTTP APIs

Routes direct incoming API requests to backend resources. Routes consist of two parts: an HTTP method and a resource path—for example, `GET /pets`. You can define specific HTTP methods for your route. Or, you can use the ANY method to match all methods that you haven't defined for a resource. You can create a `$default` route that acts as a catch-all for requests that don’t match any other routes.

Note

API Gateway decodes URL-encoded request parameters before passing them to your backend integration.

Working with path variables

You can use path variables in HTTP API routes.

For example, the `GET /pets/{petID}` route catches a `GET` request that a client submits to `https://api-id.execute-api.us-east-2.amazonaws.com/pets/6`.

A *greedy path variable* catches all child resources of a route. To create a greedy path variable, add `+` to the variable name—for example, `{proxy+}`. The greedy path variable must be at the end of the resource path.

Working with query string parameters

By default, API Gateway sends query string parameters to your backend integration if they are included in a request to an HTTP API.

For example, when a client sends a request to `https://api-id.execute-api.us-east-2.amazonaws.com/pets?id=4&type=dog`, the query string parameters `?id=4&type=dog` are sent to your integration.

Working with the `$default` route

The `$default` route catches requests that don't explicitly match other routes in your API.

When the `$default` route receives a request, API Gateway sends the full request path to the integration. For example, you can create an API with only a `$default` route and integrate it on the `ANY` method with the `https://petstore-demo-endpoint.execute-api.com` HTTP endpoint. When you send a request to `https://api-id.execute-api.us-east-2.amazonaws.com/store/checkout`, API Gateway sends a request to `https://petstore-demo-endpoint.execute-api.com/store/checkout`.

To learn more about HTTP integrations, see Working with HTTP proxy integrations for HTTP APIs.

Routing API requests

When a client sends an API request, API Gateway first determines which stage to route the request to. If the request explicitly matches a stage, API Gateway sends the request to that stage. If no stage fully matches the request, API Gateway sends the request to the `$default` stage. If there's no `$default` stage, then the API returns `{"message":"Not Found"}` and does not generate CloudWatch logs.

After selecting a stage, API Gateway selects a route. API Gateway selects the route with the most-specific match, using the following priorities:

1. Full match for a route and method.
2. Match for a route and method with a greedy path variable (`{proxy+}`).
3. The `$default` route.

If no routes match a request, API Gateway returns `{"message":"Not Found"}` to the client.

For example, consider an API with a `$default` stage and the following example routes:

1. `GET /pets/dog/1`
2. `GET /pets/dog/{id}`
3. `GET /pets/{proxy+}`
4. `ANY /{proxy+}`
5. `$default`

The following table summarizes how API Gateway routes requests to the example routes.

Request	Selected route	Explanation
`GET https://`*`api-id`*`.execute-api. `*`region`*`.amazonaws.com/pets/dog/1`	`GET /pets/dog/1`	The request fully matches this static route.
`GET https://`*`api-id`*`.execute-`	`GET /pets/dog/{id}`	The request fully matches this route.

Request	Selected route	Explanation
api.*region*.amazonaws.com/pets/dog/2		
GET https://*api-id*.execute-api.*region*.amazonaws.com/pets/cat/1	`GET /pets/{proxy+}`	The request doesn't fully match a route. The route with a GET method and a greedy path variable catches this request.
POST https://*api-id*.execute-api.*region*.amazonaws.com/test/5	`ANY /{proxy+}`	The ANY method matches all methods that you haven't defined for a route. Routes with greedy path variables have higher priority than the `$default` route.

Controlling and managing access to an HTTP API in API Gateway

API Gateway supports multiple mechanisms for controlling and managing access to your HTTP API:

- **Lambda authorizers** use Lambda functions to control access to APIs. For more information, see Working with AWS Lambda authorizers for HTTP APIs.
- **JWT authorizers** use JSON web tokens to control access to APIs. For more information, see Controlling access to HTTP APIs with JWT authorizers.
- **Standard AWS IAM roles and policies** offer flexible and robust access controls. You can use IAM roles and policies to control who can create and manage your APIs, as well as who can invoke them. For more information, see Using IAM authorization.

Working with AWS Lambda authorizers for HTTP APIs

You use a Lambda authorizer to use a Lambda function to control access to your HTTP API. Then, when a client calls your API, API Gateway invokes your Lambda function. API Gateway uses the response from your Lambda function to determine whether the client can access your API.

Payload format version

The authorizer payload format version specifies the format of the data that API Gateway sends to a Lambda authorizer, and how API Gateway interprets the response from Lambda. If you don't specify a payload format version, the AWS Management Console uses the latest version by default. If you create a Lambda authorizer by using the AWS CLI, AWS CloudFormation, or an SDK, you must specify an `authorizerPayloadFormatVersion`. The supported values are `1.0` and `2.0`.

If you need compatibility with REST APIs, use version `1.0`.

The following examples show the structure of each payload format version.

2.0

```
{
  "version": "2.0",
  "type": "REQUEST",
  "routeArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/
request",
  "identitySource": ["user1", "123"],
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": ["cookie1", "cookie2"],
  "headers": {
    "header1": "value1",
    "header2": "value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "api-id",
    "authentication": {
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "IP",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "pathParameters": { "parameter1": "value1" },
  "stageVariables": { "stageVariable1": "value1", "stageVariable2": "value2" }
}
```

1.0

```
{
  "version": "1.0",
  "type": "REQUEST",
  "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/
request",
  "identitySource": "user1,123",
  "authorizationToken": "user1,123",
  "resource": "/request",
  "path": "/request",
  "httpMethod": "GET",
  "headers": {
    "X-AMZ-Date": "20170718T062915Z",
    "Accept": "*/*",
    "HeaderAuth1": "headerValue1",
    "CloudFront-Viewer-Country": "US",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Is-Mobile-Viewer": "false",
    "User-Agent": "..."
  },
  "queryStringParameters": {
    "QueryString1": "queryValue1"
  },
  "pathParameters": {},
  "stageVariables": {
    "StageVar1": "stageValue1"
  },
  "requestContext": {
    "path": "/request",
    "accountId": "123456789012",
    "resourceId": "05c7jb",
    "stage": "test",
    "requestId": "...",
    "identity": {
      "apiKey": "...",
      "sourceIp": "...",
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "resourcePath": "/request",
    "httpMethod": "GET",
    "apiId": "abcdef123"
  }
}
```

Lambda authorizer response format

The payload format version also determines the structure of the response that you must return from your Lambda function.

Lambda function response for format 1.0

If you choose the `1.0` format version, Lambda authorizers must return an IAM policy that allows or denies access to your API route. You can use standard IAM policy syntax in the policy. For examples of IAM policies, see the section called " Control access for invoking an API". You can pass context properties to Lambda integrations or access logs by using `$context.authorizer.`*`property`*. The `context` object is optional and `claims` is a reserved placeholder and cannot be used as the context object. To learn more, see the section called "Logging variables".

Example

```
{
  "principalId": "abcdef", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/
{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

Lambda function response for format 2.0

If you choose the `2.0` format version, you can return a Boolean value or an IAM policy that uses standard IAM policy syntax from your Lambda function. To return a Boolean value, enable simple responses for the authorizer. The following examples demonstrate the format that you must code your Lambda function to return. The `context` object is optional. You can pass context properties to Lambda integrations or access logs by using `$context.authorizer.`*`property`*. To learn more, see the section called "Logging variables".

Simple response

```
{
  "isAuthorized": true/false,
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

IAM policy

```
{
  "principalId": "abcdef", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/
{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

Example Lambda authorizer functions

The following example Node.js Lambda functions demonstrate the required response formats you need to return from your Lambda function for the `2.0` payload format version.

Simple response - Node.js

```
export const handler = async(event) => {
    let response = {
        "isAuthorized": false,
        "context": {
            "stringKey": "value",
            "numberKey": 1,
            "booleanKey": true,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    };

    if (event.headers.authorization === "secretToken") {
        console.log("allowed");
        response = {
            "isAuthorized": true,
            "context": {
                "stringKey": "value",
                "numberKey": 1,
                "booleanKey": true,
                "arrayKey": ["value1", "value2"],
                "mapKey": {"value1": "value2"}
            }
        };
    }

    return response;

};
```

Simple response - Python

```
import json


def lambda_handler(event, context):
    response = {
        "isAuthorized": False,
        "context": {
            "stringKey": "value",
            "numberKey": 1,
            "booleanKey": True,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    }

    try:
        if (event["headers"]["authorization"] == "secretToken"):
            response = {
                "isAuthorized": True,
                "context": {
                    "stringKey": "value",
                    "numberKey": 1,
                    "booleanKey": True,
                    "arrayKey": ["value1", "value2"],
                    "mapKey": {"value1": "value2"}
                }
            }
            print('allowed')
            return response
        else:
            print('denied')
            return response
    except BaseException:
        print('denied')
        return response
```

IAM policy - Node.js

```
export const handler = async(event) => {
  if (event.headers.authorization == "secretToken") {
    console.log("allowed");
    return {
      "principalId": "abcdef", // The principal user identification associated with
 the token sent by the client.
      "policyDocument": {
        "Version": "2012-10-17",
        "Statement": [{
          "Action": "execute-api:Invoke",
          "Effect": "Allow",
          "Resource": event.routeArn
        }]
      },
      "context": {
        "stringKey": "value",
        "numberKey": 1,
        "booleanKey": true,
        "arrayKey": ["value1", "value2"],
        "mapKey": { "value1": "value2" }
      }
    };
  }
  else {
    console.log("denied");
    return {
      "principalId": "abcdef", // The principal user identification associated with
 the token sent by the client.
      "policyDocument": {
        "Version": "2012-10-17",
        "Statement": [{
          "Action": "execute-api:Invoke",
          "Effect": "Deny",
          "Resource": event.routeArn
        }]
      },
      "context": {
        "stringKey": "value",
        "numberKey": 1,
        "booleanKey": true,
        "arrayKey": ["value1", "value2"],
        "mapKey": { "value1": "value2" }
      }
    };
  }
};
```

IAM policy - Python

```
import json


def lambda_handler(event, context):
    response = {
        # The principal user identification associated with the token sent by
        # the client.
        "principalId": "abcdef",
        "policyDocument": {
            "Version": "2012-10-17",
            "Statement": [{
                "Action": "execute-api:Invoke",
                "Effect": "Deny",
                "Resource": event["routeArn"]
            }]
        },
        "context": {
            "stringKey": "value",
            "numberKey": 1,
            "booleanKey": True,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    }

    try:
        if (event["headers"]["authorization"] == "secretToken"):
            response = {
                # The principal user identification associated with the token
                # sent by the client.
                "principalId": "abcdef",
                "policyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [{
                        "Action": "execute-api:Invoke",
                        "Effect": "Allow",
                        "Resource": event["routeArn"]
                    }]
                },
                "context": {
                    "stringKey": "value",
                    "numberKey": 1,
                    "booleanKey": True,
                    "arrayKey": ["value1", "value2"],
                    "mapKey": {"value1": "value2"}
                }
            }
            print('allowed')
            return response
        else:
            print('denied')
            return response
    except BaseException:
        print('denied')
        return response
```

Identity sources

You can optionally specify identity sources for a Lambda authorizer. Identity sources specify the location of data that's required to authorize a request. For example, you can specify header or query string values as identity sources. If you specify identity sources, clients must include them in the request. If the client's request doesn't include the identity sources, API Gateway doesn't invoke your Lambda authorizer, and the client receives a `401` error. The following identity sources are supported:

Selection expressions

Type	Example	Notes
Header value	$request.header.*name*	Header names are case-insensitive.
Query string value	$request.querystring.*name*	Query string names are case-sensitive.
Context variable	$context.*variableName*	The value of a supported context variable.
Stage variable	$stageVariables.*variableName*	The value of a stage variable.

Caching authorizer responses

You can enable caching for a Lambda authorizer by specifying an authorizerResultTtlInSeconds. When caching is enabled for an authorizer, API Gateway uses the authorizer's identity sources as the cache key. If a client specifies the same parameters in identity sources within the configured TTL, API Gateway uses the cached authorizer result, rather than invoking your Lambda function.

To enable caching, your authorizer must have at least one identity source.

If you enable simple responses for an authorizer, the authorizer's response fully allows or denies all API requests that match the cached identity source values. For more granular permissions, disable simple responses and return an IAM policy.

By default, API Gateway uses the cached authorizer response for all routes of an API that use the authorizer. To cache responses per route, add `$context.routeKey` to your authorizer's identity sources.

Create a Lambda authorizer

When you create a Lambda authorizer, you specify the Lambda function for API Gateway to use. You must grant API Gateway permission to invoke the Lambda function by using either the function's resource policy or an IAM role. For this example, we update the resource policy for the function so that it grants API Gateway permission to invoke our Lambda function.

```
aws apigatewayv2 create-authorizer \
    --api-id abcdef123 \
    --authorizer-type REQUEST \
    --identity-source '$request.header.Authorization' \
    --name lambda-authorizer \
    --authorizer-uri 'arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-west-2:123456789012:function:my-function/invocations' \
    --authorizer-payload-format-version '2.0' \
    --enable-simple-responses
```

The following command grants API Gateway permission to invoke your Lambda function. If API Gateway doesn't have permission to invoke your function, clients receive a `500 Internal Server Error`.

```
aws lambda add-permission \
    --function-name my-authorizer-function \
    --statement-id apigateway-invoke-permissions-abc123 \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-
id/authorizers/authorizer-id"
```

After you've created an authorizer and granted API Gateway permission to invoke it, update your route to use the authorizer.

```
aws apigatewayv2 update-route \
    --api-id abcdef123 \
    --route-id acd123 \
    --authorization-type CUSTOM \
    --authorizer-id def123
```

Troubleshooting Lambda authorizers

If API Gateway can't invoke your Lambda authorizer, or your Lambda authorizer returns a response in an invalid format, clients receive a `500 Internal Server Error`.

To troubleshoot errors, enable access logging for your API stage. Include the `$context.authorizer.error` logging variable in your log format.

If the logs indicate that API Gateway doesn't have permission to invoke your function, update your function's resource policy or provide an IAM role to grant API Gateway permission to invoke your authorizer.

If the logs indicate that your Lambda function returns an invalid response, verify that your Lambda function returns a response in the required format.

Controlling access to HTTP APIs with JWT authorizers

You can use JSON Web Tokens (JWTs) as a part of OpenID Connect (OIDC) and OAuth 2.0 frameworks to restrict client access to your APIs.

If you configure a JWT authorizer for a route of your API, API Gateway validates the JWTs that clients submit with API requests. API Gateway allows or denies requests based on token validation, and optionally, scopes in the token. If you configure scopes for a route, the token must include at least one of the route's scopes.

You can configure distinct authorizers for each route of an API, or use the same authorizer for multiple routes.

> **Note**
>
> There is no standard mechanism to differentiate JWT access tokens from other types of JWTs, such as OpenID Connect ID tokens. Unless you require ID tokens for API authorization, we recommend that you configure your routes to require authorization scopes. You can also configure your JWT authorizers to require issuers or audiences that your identity provider uses only when issuing JWT access tokens.

Authorizing API requests with a JWT authorizer

API Gateway uses the following general workflow to authorize requests to routes that are configured to use a JWT authorizer.

1. Check the identitySource for a token. The identitySource can include only the token, or the token prefixed with Bearer.
2. Decode the token.
3. Check the token's algorithm and signature by using the public key that is fetched from the issuer's jwks_uri. Currently, only RSA-based algorithms are supported. API Gateway can cache the public key for two hours. As a best practice, when you rotate keys, allow a grace period during which both the old and new keys are valid.
4. Validate claims. API Gateway evaluates the following token claims:
   - kid – The token must have a header claim that matches the key in the jwks_uri that signed the token.
   - iss – Must match the issuer that is configured for the authorizer.
   - aud or client_id – Must match one of the audience entries that is configured for the authorizer. API Gateway validates client_id only if aud is not present. When both aud and client_id are present, API Gateway evaluates aud.
   - exp – Must be after the current time in UTC.
   - nbf – Must be before the current time in UTC.
   - iat – Must be before the current time in UTC.
   - scope or scp – The token must include at least one of the scopes in the route's authorizationScopes.

If any of these steps fail, API Gateway denies the API request.

After validating the JWT, API Gateway passes the claims in the token to the API route’s integration. Backend resources, such as Lambda functions, can access the JWT claims. For example, if the JWT includes an identity claim emailID, it's available to a Lambda integration in $event.requestContext.authorizer.jwt.claims.emailID. For more information about the payload that API Gateway sends to Lambda integrations, see the section called “AWS Lambda integrations”.

Create a JWT authorizer

Before you create a JWT authorizer, you must register a client application with an identity provider. You must also have created an HTTP API. For examples of creating an HTTP API, see Creating an HTTP API.

Create a JWT authorizer using the console

The following steps show how to create JWT authorizer using the console.

To create a JWT authorizer using the console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an HTTP API.
3. In the main navigation pane, choose **Authorization**.
4. Choose the **Manage authorizers** tab.
5. Choose **Create**.
6. For **Authorizer type**, choose **JWT**.
7. Configure your JWT authorizer, and specify an **Identity source** that defines the source of the token.
8. Choose **Create**.

Create a JWT authorizer using the AWS CLI

The following AWS CLI command creates a JWT authorizer. For `jwt-configuration`, specify the `Audience` and `Issuer` for your identity provider. If you use Amazon Cognito as an identity provider, the `IssuerUrl` is `https://cognito-idp.`*`us-east-2`*`.amazonaws.com/`*`userPoolID`*.

```
aws apigatewayv2 create-authorizer \
    --name authorizer-name \
    --api-id api-id \
    --authorizer-type JWT \
    --identity-source '$request.header.Authorization' \
    --jwt-configuration Audience=audience,Issuer=IssuerUrl
```

Create a JWT authorizer using AWS CloudFormation

The following AWS CloudFormation template creates an HTTP API with a JWT authorizer that uses Amazon Cognito as an identity provider.

The output of the AWS CloudFormation template is a URL for an Amazon Cognito hosted UI where clients can sign up and sign in to receive a JWT. After a client signs in, the client is redirected to your HTTP API with an access token in the URL. To invoke the API with the access token, change the # in the URL to a ? to use the token as a query string parameter.

Example AWS CloudFormation template

```
AWSTemplateFormatVersion: '2010-09-09'
Description: |
  Example HTTP API with a JWT authorizer. This template includes an Amazon Cognito user
 pool as the issuer for the JWT authorizer
  and an Amazon Cognito app client as the audience for the authorizer. The outputs
 include a URL for an Amazon Cognito hosted UI where clients can
  sign up and sign in to receive a JWT. After a client signs in, the client is
 redirected to your HTTP API with an access token
  in the URL. To invoke the API with the access token, change the '#' in the URL to a
 '?' to use the token as a query string parameter.

Resources:
  MyAPI:
    Type: AWS::ApiGatewayV2::Api
    Properties:
      Description: Example HTTP API
      Name: api-with-auth
      ProtocolType: HTTP
      Target: !GetAtt MyLambdaFunction.Arn
  DefaultRouteOverrides:
    Type: AWS::ApiGatewayV2::ApiGatewayManagedOverrides
    Properties:
      ApiId: !Ref MyAPI
      Route:
        AuthorizationType: JWT
        AuthorizerId: !Ref JWTAuthorizer
  JWTAuthorizer:
    Type: AWS::ApiGatewayV2::Authorizer
    Properties:
      ApiId: !Ref MyAPI
      AuthorizerType: JWT
      IdentitySource:
        - '$request.querystring.access_token'
      JwtConfiguration:
        Audience:
        - !Ref AppClient
        Issuer: !Sub https://cognito-idp.${AWS::Region}.amazonaws.com/${UserPool}
      Name: test-jwt-authorizer
  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: nodejs18.x
      Role: !GetAtt FunctionExecutionRole.Arn
      Handler: index.handler
      Code:
        ZipFile: |
          exports.handler = async (event) => {
              const response = {
                  statusCode: 200,
                  body: JSON.stringify('Hello from the ' + event.routeKey + ' route!'),
              };
              return response;
          };
  APIInvokeLambdaPermission:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyLambdaFunction
      Action: lambda:InvokeFunction
      Principal: apigateway.amazonaws.com
      SourceArn: !Sub arn:${AWS::Partition}:execute-api:${AWS::Region}:
${AWS::AccountId}:${MyAPI}/$default/$default
  FunctionExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
              - lambda.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
  UserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      UserPoolName: http-api-user-pool
      AutoVerifiedAttributes:
        - email
      Schema:
        - Name: name
```

```
          AttributeDataType: String
          Mutable: true
          Required: true
        - Name: email
          AttributeDataType: String
          Mutable: false
          Required: true
  AppClient:
    Type: AWS::Cognito::UserPoolClient
    Properties:
      AllowedOAuthFlows:
        - implicit
      AllowedOAuthScopes:
        - aws.cognito.signin.user.admin
        - email
        - openid
        - profile
      AllowedOAuthFlowsUserPoolClient: true
      ClientName: api-app-client
      CallbackURLs:
        - !Sub https://${MyAPI}.execute-api.${AWS::Region}.amazonaws.com
      ExplicitAuthFlows:
        - ALLOW_USER_PASSWORD_AUTH
        - ALLOW_REFRESH_TOKEN_AUTH
      UserPoolId: !Ref UserPool
      SupportedIdentityProviders:
        - COGNITO
  HostedUI:
    Type: AWS::Cognito::UserPoolDomain
    Properties:
      Domain: !Join
        - '-'
        - - !Ref MyAPI
          - !Ref AppClient
      UserPoolId: !Ref UserPool
Outputs:
  SignupURL:
    Value: !Sub https://${HostedUI}.auth.${AWS::Region}.amazoncognito.com/login?
client_id=${AppClient}&response_type=token&scope=email+profile&redirect_uri=https://
${MyAPI}.execute-api.${AWS::Region}.amazonaws.com
```

Update a route to use a JWT authorizer

You can use the console, the AWS CLI, or an AWS SDK to update a route to use a JWT authorizer.

Update a route to use a JWT authorizer by using the console

The following steps show how to update a route to use JWT authorizer using the console.

To create a JWT authorizer using the console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an HTTP API.
3. In the main navigation pane, choose **Authorization**.
4. Choose a method, and then select your authorizer from the dropdown menu, and choose **Attach authorizer**.

Update a route to use a JWT authorizer by using the AWS CLI

The following command updates a route to use a JWT authorizer using the AWS CLI.

```
aws apigatewayv2 update-route \
   --api-id api-id  \
   --route-id route-id  \
   --authorization-type JWT \
   --authorizer-id authorizer-id \
   --authorization-scopes user.email
```

Using IAM authorization

You can enable IAM authorization for HTTP API routes. When IAM authorization is enabled, clients must use Signature Version 4 to sign their requests with AWS credentials. API Gateway invokes your API route only if the client has `execute-api` permission for the route.

IAM authorization for HTTP APIs is similar to that for REST APIs.

> **Note**
>
> Resource policies aren't currently supported for HTTP APIs.

For examples of IAM policies that grant clients the permission to invoke APIs, see the section called “ Control access for invoking an API”.

Enable IAM authorization for a route

The following AWS CLI command enables IAM authorization for an HTTP API route.

```
aws apigatewayv2 update-route \
    --api-id abc123 \
    --route-id abcdef \
    --authorization-type AWS_IAM
```

Configuring integrations for HTTP APIs

Integrations connect a route to backend resources. HTTP APIs support Lambda proxy, AWS service, and HTTP proxy integrations. For example, you can configure a `POST` request to the `/signup` route of your API to integrate with a Lambda function that handles signing up customers.

Topics

Working with AWS Lambda proxy integrations for HTTP APIs

A Lambda proxy integration enables you to integrate an API route with a Lambda function. When a client calls your API, API Gateway sends the request to the Lambda function and returns the function's response to the client. For examples of creating an HTTP API, see Creating an HTTP API.

Payload format version

The payload format version specifies the format of the event that API Gateway sends to a Lambda integration, and how API Gateway interprets the response from Lambda. If you don't specify a payload format version, the AWS Management Console uses the latest version by default. If you create a Lambda integration by using the AWS CLI, AWS CloudFormation, or an SDK, you must specify a `payloadFormatVersion`. The supported values are `1.0` and `2.0`.

For more information about how to set the `payloadFormatVersion`, see create-integration. For more information about how to determine the `payloadFormatVersion` of an existing integration, see get-integration.

The following examples show the structure of each payload format version.

Note

Header names are lowercased.

Format `2.0` doesn't have `multiValueHeaders` or `multiValueQueryStringParameters` fields. Duplicate headers are combined with commas and included in the `headers` field. Duplicate query strings are combined with commas and included in the `queryStringParameters` field.

Format `2.0` includes a new `cookies` field. All cookie headers in the request are combined with commas and added to the `cookies` field. In the response to the client, each cookie becomes a `set-cookie` header.

2.0

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "api-id",
    "authentication": {
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "authorizer": {
      "jwt": {
        "claims": {
          "claim1": "value1",
          "claim2": "value2"
        },
        "scopes": [
          "scope1",
          "scope2"
        ]
      }
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "192.0.2.1",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from Lambda",
  "pathParameters": {
    "parameter1": "value1"
  },
  "isBase64Encoded": false,
  "stageVariables": {
    "stageVariable1": "value1",
    "stageVariable2": "value2"
  }
```

```
}
```

1.0

```
{
  "version": "1.0",
  "resource": "/my/path",
  "path": "/my/path",
  "httpMethod": "GET",
  "headers": {
    "header1": "value1",
    "header2": "value2"
  },
  "multiValueHeaders": {
    "header1": [
      "value1"
    ],
    "header2": [
      "value1",
      "value2"
    ]
  },
  "queryStringParameters": {
    "parameter1": "value1",
    "parameter2": "value"
  },
  "multiValueQueryStringParameters": {
    "parameter1": [
      "value1",
      "value2"
    ],
    "parameter2": [
      "value"
    ]
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "id",
    "authorizer": {
      "claims": null,
      "scopes": null
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "extendedRequestId": "request-id",
    "httpMethod": "GET",
    "identity": {
      "accessKey": null,
      "accountId": null,
      "caller": null,
      "cognitoAuthenticationProvider": null,
      "cognitoAuthenticationType": null,
      "cognitoIdentityId": null,
      "cognitoIdentityPoolId": null,
      "principalOrgId": null,
      "sourceIp": "192.0.2.1",
      "user": null,
      "userAgent": "user-agent",
      "userArn": null,
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "path": "/my/path",
    "protocol": "HTTP/1.1",
    "requestId": "id=",
    "requestTime": "04/Mar/2020:19:15:17 +0000",
    "requestTimeEpoch": 1583349317135,
    "resourceId": null,
    "resourcePath": "/my/path",
    "stage": "$default"
  },
  "pathParameters": null,
  "stageVariables": null,
  "body": "Hello from Lambda!",
  "isBase64Encoded": false
}
```

Lambda function response format

The payload format version determines the structure of the response that your Lambda function must return.

Lambda function response for format 1.0

With the `1.0` format version, Lambda integrations must return a response in the following JSON format:

Example

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headername": "headervalue", ... },
    "multiValueHeaders": { "headername": ["headervalue", "headervalue2", ...], ... },
    "body": "..."
}
```

Lambda function response for format 2.0

With the `2.0` format version, API Gateway can infer the response format for you. API Gateway makes the following assumptions if your Lambda function returns valid JSON and doesn't return a `statusCode`:

- `isBase64Encoded` is `false`.
- `statusCode` is `200`.
- `content-type` is `application/json`.
- body is the function's response.

The following examples show the output of a Lambda function and API Gateway's interpretation.

Lambda function output	API Gateway interpretation

`"Hello from Lambda!"`	`{`
	`  "isBase64Encoded": false,`
	`  "statusCode": 200,`
	`  "body": "Hello from Lambda!",`
	`  "headers": {`

Lambda function output	API Gateway interpretation

	`    "content-type": "application/json"`
	`  }`
	`}`

`{ "message": "Hello from Lambda!" }`	`{`
	`  "isBase64Encoded": false,`
	`  "statusCode": 200,`
	`  "body": "{ \"message\": \"Hello from Lambda!\" }",`
	`  "headers": {`
	`    "content-type": "application/json"`
	`  }`
	`}`

To customize the response, your Lambda function should return a response with the following format.

```
{
    "cookies" : ["cookie1", "cookie2"],
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headername": "headervalue", ... },
    "body": "Hello from Lambda!"
}
```

Working with HTTP proxy integrations for HTTP APIs

An HTTP proxy integration enables you to connect an API route to a publicly routable HTTP endpoint. With this integration type, API Gateway passes the entire request and response between the frontend and the backend.

To create an HTTP proxy integration, provide the URL of a publicly routable HTTP endpoint.

HTTP proxy integration with path variables

You can use path variables in HTTP API routes.

For example, the route `/pets/{petID}` catches requests to `/pets/6`. You can reference path variables in the integration URI to send the variable's contents to an integration. An example is `/pets/extendedpath/{petID}`.

You can use greedy path variables to catch all child resources of a route. To create a greedy path variable, add + to the variable name—for example, `{proxy+}`.

To set up a route with an HTTP proxy integration that catches all requests, create an API route with a greedy path variable (for example, `/parent/{proxy+}`). Integrate the route with an HTTP endpoint (for example, `https://petstore-demo-endpoint.execute-api.com/petstore/{proxy}`) on the ANY method. The greedy path variable must be at the end of the resource path.

Working with AWS service integrations for HTTP APIs

You can integrate your HTTP API with AWS services by using *first-class integrations*. A first-class integration connects an HTTP API route to an AWS service API. When a client invokes a route that's backed by a first-class integration, API Gateway invokes an AWS service API for you. For example, you can use first-class integrations to send a message to an Amazon Simple Queue Service queue, or to start an AWS Step Functions state machine. For supported service actions, see the section called "AWS service integrations reference".

Mapping request parameters

First-class integrations have required and optional parameters. You must configure all required parameters to create an integration. You can use static values or map parameters that are dynamically evaluated at runtime. For a full list of supported integrations and parameters, see the section called "AWS service integrations reference".

Parameter mapping

Type	Example	Notes
Header value	$request.header.*name*	Header names are case-insensitive. API Gateway combines multiple header values with commas, for example `"header1": "value1,value2"` .

Type	Example	Notes
Query string value	$request.querystring.*name*	Query string names are case-sensitive. API Gateway combines multiple values with commas, for example `"querystring1": "Value1,Value2"` .
Path parameter	$request.path.*name*	The value of a path parameter in the request. For example if the route is `/pets/{petId}` , you can map the `petId` parameter from the request with *`$request.path.petId`* .
Request body passthrough	$request.body	API Gateway passes the entire request body through.

Type	Example	Notes
Request body	$request.body.*name*	A JSON path expression. Recursive descent (`$request.body..` *`name`*) and filter expressions (`?(`*`expression`*`)`) aren't supported. **Note** When you specify a JSON path, API Gateway truncates the request body at 100 KB and then applies the selection expression. To send payloads larger than 100 KB, specify `$request.body` .
Context variable	$context.*variableName*	The value of a supported context variable.
Stage variable	$stageVariables.*variableName*	The value of a stage variable.
Static value	*string*	A constant value.

Create a first-class integration

Before you create a first-class integration, you must create an IAM role that grants API Gateway permissions to invoke the AWS service action that you're integrating with. To learn more, see Creating a role for an AWS service.

To create a first-class integration, choose a supported AWS service action, such as SQS-SendMessage, configure the request parameters, and provide a role that grants API Gateway permissions to invoke the integrated AWS service API. Depending on the integration subtype, different request parameters are required. To learn more, see the section called “AWS service integrations reference”.

The following AWS CLI command creates an integration that sends an Amazon SQS message.

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-subtype SQS-SendMessage \
    --integration-type AWS_PROXY \
    --payload-format-version 1.0 \
    --credentials-arn arn:aws:iam::123456789012:role/apigateway-sqs \
    --request-parameters '{"QueueUrl": "$request.header.queueUrl", "MessageBody":
 "$request.body.message"}'
```

Create a first-class integration using AWS CloudFormation

The following example shows an AWS CloudFormation snippet that creates a /{source}/{detailType} route with a first-class integration with Amazon EventBridge.

The Source parameter is mapped to the {source} path parameter, the DetailType is mapped to the {DetailType} path parameter, and the Detail parameter is mapped to the request body.

The snippet does not show the event bus or the IAM role that grants API Gateway permissions to invoke the PutEvents action.

```
Route:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref HttpApi
      AuthorizationType: None
      RouteKey: 'POST /{source}/{detailType}'
      Target: !Join
        - /
        - - integrations
          - !Ref Integration
  Integration:
    Type: AWS::ApiGatewayV2::Integration
    Properties:
      ApiId: !Ref HttpApi
      IntegrationType: AWS_PROXY
      IntegrationSubtype: EventBridge-PutEvents
      CredentialsArn: !GetAtt EventBridgeRole.Arn
      RequestParameters:
        Source: $request.path.source
        DetailType: $request.path.detailType
        Detail: $request.body
        EventBusName: !GetAtt EventBus.Arn
      PayloadFormatVersion: "1.0"
```

Integration subtype reference

The following integration subtypes are supported for HTTP APIs.

Integration subtypes

- EventBridge-PutEvents
- SQS-SendMessage
- SQS-ReceiveMessage
- SQS-DeleteMessage
- SQS-PurgeQueue
- AppConfig-GetConfiguration
- Kinesis-PutRecord
- StepFunctions-StartExecution
- StepFunctions-StartSyncExecution
- StepFunctions-StopExecution

EventBridge-PutEvents

Sends custom events to Amazon EventBridge so that they can be matched to rules.

EventBridge-PutEvents 1.0

Parameter	Required
Detail	True
DetailType	True

Parameter	**Required**
Source	True
Time	False
EventBusName	False
Resources	False
Region	False
TraceHeader	False

To learn more, see PutEvents in the *Amazon EventBridge API Reference*.

SQS-SendMessage

Delivers a message to the specified queue.

SQS-SendMessage 1.0

Parameter	**Required**
QueueUrl	True
MessageBody	True
DelaySeconds	False
MessageAttributes	False
MessageDeduplicationId	False
MessageGroupId	False
MessageSystemAttributes	False
Region	False

To learn more, see SendMessage in the *Amazon Simple Queue Service API Reference*.

SQS-ReceiveMessage

Retrieves one or more messages (up to 10), from the specified queue.

SQS-ReceiveMessage 1.0

Parameter	Required
QueueUrl	True
AttributeNames	False
MaxNumberOfMessages	False
MessageAttributeNames	False
ReceiveRequestAttemptId	False
VisibilityTimeout	False
WaitTimeSeconds	False
Region	False

To learn more, see ReceiveMessage in the *Amazon Simple Queue Service API Reference*.

SQS-DeleteMessage

Deletes the specified message from the specified queue.

SQS-DeleteMessage 1.0

Parameter	Required
ReceiptHandle	True
QueueUrl	True
Region	False

To learn more, see DeleteMessage in the *Amazon Simple Queue Service API Reference*.

SQS-PurgeQueue

Deletes all messages in the specified queue.

SQS-PurgeQueue 1.0

Parameter	Required
QueueUrl	True
Region	False

To learn more, see PurgeQueue in the *Amazon Simple Queue Service API Reference*.

AppConfig-GetConfiguration

Receive information about a configuration.

AppConfig-GetConfiguration 1.0

Parameter	Required
Application	True
Environment	True
Configuration	True
ClientId	True
ClientConfigurationVersion	False
Region	False

To learn more, see GetConfiguration in the *AWS AppConfig API Reference*.

Kinesis-PutRecord

Writes a single data record into an Amazon Kinesis data stream.

Kinesis-PutRecord 1.0

Parameter	Required
StreamName	True
Data	True
PartitionKey	True
SequenceNumberForOrdering	False
ExplicitHashKey	False
Region	False

To learn more, see PutRecord in the *Amazon Kinesis Data Streams API Reference*.

StepFunctions-StartExecution

Starts a state machine execution.

StepFunctions-StartExecution 1.0

Parameter	Required
StateMachineArn	True
Name	False
Input	False
Region	False

To learn more, see StartExecution in the *AWS Step Functions API Reference*.

StepFunctions-StartSyncExecution

Starts a synchronous state machine execution.

StepFunctions-StartSyncExecution 1.0

Parameter	Required
StateMachineArn	True
Name	False
Input	False
Region	False
TraceHeader	False

To learn more, see StartSyncExecution in the *AWS Step Functions API Reference*.

StepFunctions-StopExecution

Stops an execution.

StepFunctions-StopExecution 1.0

Parameter	Required
ExecutionArn	True
Cause	False
Error	False
Region	False

To learn more, see StopExecution in the *AWS Step Functions API Reference*.

Working with private integrations for HTTP APIs

Private integrations enable you to create API integrations with private resources in a VPC, such as Application Load Balancers or Amazon ECS container-based applications.

You can expose your resources in a VPC for access by clients outside of the VPC by using private integrations. You can control access to your API by using any of the authorization methods that API Gateway supports.

To create a private integration, you must first create a VPC link. To learn more about VPC links, see Working with VPC links for HTTP APIs.

After you've created a VPC link, you can set up private integrations that connect to an Application Load Balancer, Network Load Balancer, or resources registered with an AWS Cloud Map service.

To create a private integration, all resources must be owned by the same AWS account (including the load balancer or AWS Cloud Map service, VPC link and HTTP API).

By default, private integration traffic uses the HTTP protocol. You can specify a `tlsConfig` if you require private integration traffic to use HTTPS.

> **Note**
>
> For private integrations, API Gateway includes the stage portion of the API endpoint in the request to your backend resources. For example, a request to the `test` stage of an API includes `test/route-path` in the request to your private integration. To remove the stage name from the request to your backend resources, use parameter mapping to overwrite the request path to `$request.path`.

Create a private integration using an Application Load Balancer or Network Load Balancer

Before you create a private integration, you must create a VPC link. To learn more about VPC links, see Working with VPC links for HTTP APIs.

To create a private integration with an Application Load Balancer or Network Load Balancer, create an HTTP proxy integration, specify the VPC link to use, and provide the listener ARN of the load balancer.

Use the following command to create a private integration that connects to a load balancer by using a VPC link.

```
aws apigatewayv2 create-integration --api-id api-id --integration-type HTTP_PROXY \
    --integration-method GET --connection-type VPC_LINK \
    --connection-id VPC-link-ID \
    --integration-uri arn:aws:elasticloadbalancing:us-east-2:123456789012:listener/app/
my-load-balancer/50dc6c495c0c9188/0467ef3c8400ae65
    --payload-format-version 1.0
```

Create a private integration using AWS Cloud Map service discovery

Before you create a private integration, you must create a VPC link. To learn more about VPC links, see Working with VPC links for HTTP APIs.

For integrations with AWS Cloud Map, API Gateway uses `DiscoverInstances` to identify resources. You can use query parameters to target specific resources. The registered resources' attributes must include IP addresses and ports. API Gateway distributes requests across healthy resources that are returned from `DiscoverInstances`. To learn more, see DiscoverInstances in the AWS Cloud Map API Reference.

> **Note**
>
> If you use Amazon ECS to populate entries in AWS Cloud Map, you must configure your Amazon ECS task to use SRV records with Amazon ECS Service Discovery or turn on Amazon ECS Service Connect. For more information, see Interconnecting services in the Amazon Elastic Container Service Developer Guide.

To create a private integration with AWS Cloud Map, create an HTTP proxy integration, specify the VPC link to use, and provide the ARN of the AWS Cloud Map service.

Use the following command to create a private integration that uses AWS Cloud Map service discovery to identify resources.

```
aws apigatewayv2 create-integration --api-id api-id --integration-type HTTP_PROXY  \
    --integration-method GET --connection-type VPC_LINK \
    --connection-id VPC-link-ID \
    --integration-uri arn:aws:servicediscovery:us-east-2:123456789012:service/srv-id?
stage=prod&deployment=green_deployment
    --payload-format-version 1.0
```

Working with VPC links for HTTP APIs

VPC links enable you to create private integrations that connect your HTTP API routes to private resources in a VPC, such as Application Load Balancers or Amazon ECS container-based applications. To learn more about creating private integrations, see Working with private integrations for HTTP APIs.

A private integration uses a VPC link to encapsulate connections between API Gateway and targeted VPC resources. You can reuse VPC links across different routes and APIs.

When you create a VPC link, API Gateway creates and manages elastic network interfaces for the VPC link in your account. This process can take a few minutes. When a VPC link is ready to use, its state transitions from `PENDING` to `AVAILABLE`.

Note

If no traffic is sent over the VPC link for 60 days, it becomes `INACTIVE`. When a VPC link is in an `INACTIVE` state, API Gateway deletes all of the VPC link's network interfaces. This causes API requests that depend on the VPC link to fail. If API requests resume, API Gateway reprovisions network interfaces. It can take a few minutes to create the network interfaces and reactivate the VPC link. You can use the VPC link status to monitor the state of your VPC link.

Create a VPC link by using the AWS CLI

Use the following command to create a VPC link. To create a VPC link, all resources involved must be owned by the same AWS account.

```
aws apigatewayv2 create-vpc-link --name MyVpcLink \
    --subnet-ids subnet-aaaa subnet-bbbb \
    --security-group-ids sg1234 sg5678
```

Note

VPC links are immutable. After you create a VPC link, you can't change its subnets or security groups.

Delete a VPC link by using the AWS CLI

Use the following command to delete a VPC link.

```
aws apigatewayv2 delete-vpc-link --vpc-link-id abcd123
```

Availability by Region

VPC links for HTTP APIs are supported in the following Regions and Availability Zones:

Region name	Region	Supported Availability Zones
US East (Ohio)	us-east-2	use2-az1, use2-az2, use2-az3
US East (N. Virginia)	us-east-1	use1-az1, use1-az2, use1-az4, use1-az5, use1-az6
US West (N. California)	us-west-1	usw1-az1, usw1-az3
US West (Oregon)	us-west-2	usw2-az1, usw2-az2, usw2-az3, usw2-az4
Asia Pacific (Hong Kong)	ap-east-1	ape1-az2, ape1-az3
Asia Pacific (Mumbai)	ap-south-1	aps1-az1, aps1-az2, aps1-az3
Asia Pacific (Seoul)	ap-northeast-2	apne2-az1, apne2-az2, apne2-az3
Asia Pacific (Singapore)	ap-southeast-1	apse1-az1, apse1-az2, apse1-az3
Asia Pacific (Sydney)	ap-southeast-2	apse2-az1, apse2-az2, apse2-az3
Asia Pacific (Tokyo)	ap-northeast-1	apne1-az1, apne1-az2, apne1-az4
Canada (Central)	ca-central-1	cac1-az1, cac1-az2

Region name	Region	Supported Availability Zones
Europe (Frankfurt)	eu-central-1	euc1-az1, euc1-az2, euc1-az3
Europe (Ireland)	eu-west-1	euw1-az1, euw1-az2, euw1-az3
Europe (London)	eu-west-2	euw2-az1, euw2-az2, euw2-az3
Europe (Paris)	eu-west-3	euw3-az1, euw3-az3
Europe (Stockholm)	eu-north-1	eun1-az1, eun1-az2, eun1-az3
Middle East (Bahrain)	me-south-1	mes1-az1, mes1-az2, mes1-az3
South America (São Paulo)	sa-east-1	sae1-az1, sae1-az2, sae1-az3
AWS GovCloud (US-West)	us-gov-west-1	usgw1-az1, usgw1-az2, usgw1-az3

Configuring CORS for an HTTP API

Cross-origin resource sharing (CORS) is a browser security feature that restricts HTTP requests that are initiated from scripts running in the browser. If you cannot access your API and receive an error message that contains `Cross-Origin Request Blocked`, you might need to enable CORS.

CORS is typically required to build web applications that access APIs hosted on a different domain or origin. You can enable CORS to allow requests to your API from a web application hosted on a different domain. For example, if your API is hosted on `https://{api_id}.execute-api.`

`{region}.amazonaws.com/` and you want to call your API from a web application hosted on `example.com`, your API must support CORS.

If you configure CORS for an API, API Gateway automatically sends a response to preflight OPTIONS requests, even if there isn't an OPTIONS route configured for your API. For a CORS request, API Gateway adds the configured CORS headers to the response from an integration.

> **Note**
>
> If you configure CORS for an API, API Gateway ignores CORS headers returned from your backend integration.

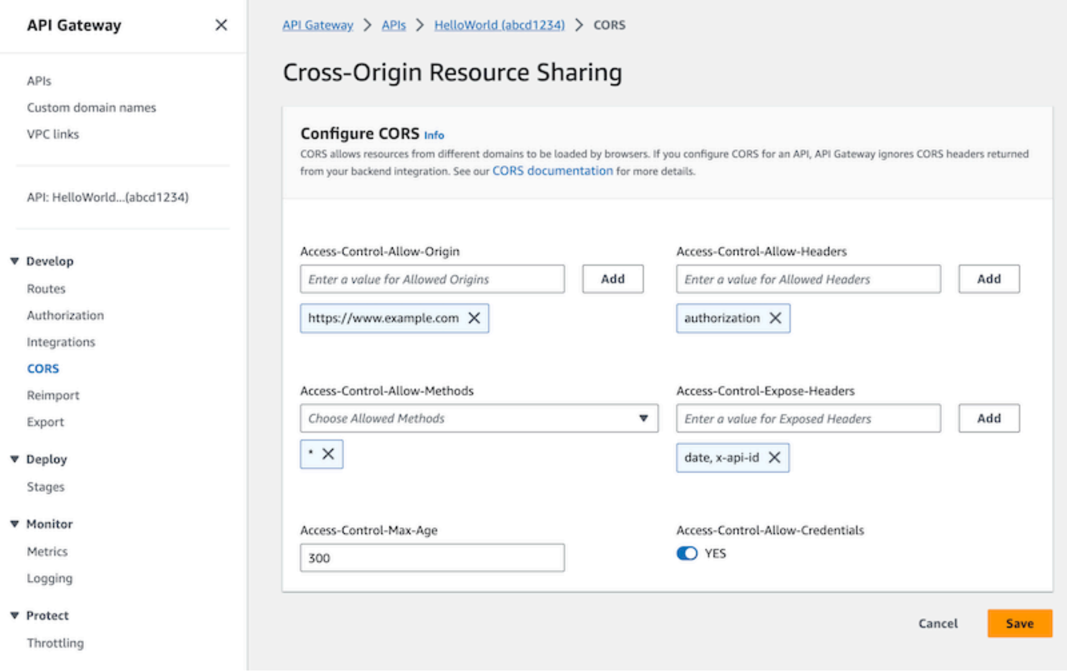
You can specify the following parameters in a CORS configuration. To add these parameters using the API Gateway HTTP API console, choose **Add** after you enter your value.

CORS headers	CORS configuration property	Example values
Access-Control-Allow-Origin	allowOrigins	• `https://www.example.com` • * (allow all origins) • `https://*` (allow any origin that begins with https://) • `http://*` (allow any origin that begins with http://)
Access-Control-Allow-Credentials	allowCredentials	true
Access-Control-Expose-Headers	exposeHeaders	Date, x-api-id
Access-Control-Max-Age	maxAge	300
Access-Control-Allow-Methods	allowMethods	GET, POST, DELETE, *

CORS headers	CORS configuration property	Example values
Access-Control-Allow-Headers	allowHeaders	Authorization, *

To return CORS headers, your request must contain an `origin` header.

Your CORS configuration might look similar to the following:



Cross-Origin Resource Sharing

Configure CORS

CORS allows resources from different domains to be loaded by browsers. If you configure CORS for an API, API Gateway ignores CORS headers returned from your backend integration. See our CORS documentation for more details.

Access-Control-Allow-Origin: https://www.example.com

Access-Control-Allow-Headers: authorization

Access-Control-Allow-Methods: *

Access-Control-Expose-Headers: date, x-api-id

Access-Control-Max-Age: 300

Access-Control-Allow-Credentials: YES

Configuring CORS for an HTTP API with a `$default` route and JWT authorizer

You can enable CORS and configure authorization for any route of an HTTP API. When you enable CORS and authorization for the [`$default` route](#), there are some special considerations. The `$default` route catches requests for all methods and routes that you haven't explicitly defined, including `OPTIONS` requests. To support unauthorized `OPTIONS` requests, add an `OPTIONS /{proxy+}` route to your API that doesn't require authorization and attach an integration to the route. The `OPTIONS /{proxy+}` route has higher priority than the `$default` route. As a result, it enables clients to submit `OPTIONS` requests to your API without authorization. For more information about routing priorities, see [Routing API requests](#).

Configure CORS for an HTTP API by using the AWS CLI

You can use the following command to enable CORS requests from `https://www.example.com`.

Example

```
aws apigatewayv2 update-api --api-id api-id --cors-configuration AllowOrigins="https://
www.example.com"
```

For more information, see CORS in the Amazon API Gateway Version 2 API Reference.

Transforming API requests and responses

You can modify API requests from clients before they reach your backend integrations. You can also change the response from integrations before API Gateway returns the response to clients. You use *parameter mapping* to modify API requests and responses for HTTP APIs. To use parameter mapping, you specify API request or response parameters to modify, and specify how to modify those parameters.

Transforming API requests

You use request parameters to change requests before they reach your backend integrations. You can modify headers, query strings, or the request path.

Request parameters are a key-value map. The key identifies the location of the request parameter to change, and how to change it. The value specifies the new data for the parameter.

The following table shows supported keys.

Parameter mapping keys

Type	Syntax
Header	`append|overwrite|remove:header.` *`headername`*
Query string	`append|overwrite|remove:querystring.` *`querystring-name`*
Path	`overwrite:path`

The following table shows supported values that you can map to parameters.

Request parameter mapping values

Type	Syntax	Notes
Header value	$request.header.*name* or ${request.header.*name*}	Header names are case-insensitive. API Gateway combines multiple header values with commas, for example `"header1": "value1,value2"` . Some headers are reserved. To learn more, see the section called "Reserved headers".
Query string value	$request.querystring.*name* or ${request.querystring.*name*}	Query string names are case-sensitive. API Gateway combines multiple values with commas, for example `"querystring1" "Value1,Value2"` .
Request body	$request.body.*name* or ${request.body.*name*}	A JSON path expressio n. Recursive descent (`$request.body..name`) and filter expressions (`?(expression)`) aren't supported. **Note** When you specify a JSON path, API Gateway truncates the request body at 100 KB and then applies the selection expression. To send payloads larger

Type	Syntax	Notes
		than 100 KB, specify `$request.body` .
Request path	$request.path or ${request .path}	The request path, without the stage name.
Path parameter	$request.path.*name* or ${request.path.*name*}	The value of a path parameter in the request. For example if the route is `/pets/{petId}` , you can map the `petId` parameter from the request with `$request.path.petId` .
Context variable	$context.*variableName* or ${context.*variableName* }	The value of a context variable. **Note** Only the special characters `.` and _ are supported.
Stage variable	$stageVariables.*variableName* or ${stageVariables.*variableName* }	The value of a stage variable.
Static value	*string*	A constant value.

Note

To use multiple variables in a selection expression, enclose the variable in brackets. For example, `${request.path.name} ${request.path.id}`.

Transforming API responses

You use response parameters to transform the HTTP response from a backend integration before returning the response to clients. You can modify headers or the status code of a response before API Gateway returns the response to clients.

You configure response parameters for each status code that your integration returns. Response parameters are a key-value map. The key identifies the location of the request parameter to change, and how to change it. The value specifies the new data for the parameter.

The following table shows supported keys.

Response parameter mapping keys

Type	Syntax
Header	`append|overwrite|remove:header.` *headername*
Status code	`overwrite:statuscode`

The following table shows supported values that you can map to parameters.

Response parameter mapping values

Type	Syntax	Notes
Header value	$response.header.*name* or ${response.header.*name*}	Header names are case-insensitive. API Gateway combines multiple header values with commas, for example `"header1": "value1,value2"` . Some headers are reserved. To learn more, see the section called "Reserved headers".
Response body	$response.body.*name* or ${response.body.*name*}	A JSON path expression. Recursive descent (`$response.body..na`

Type	Syntax	Notes
		`me`) and filter expressions (`?(expression)`) aren't supported. **Note** When you specify a JSON path, API Gateway truncates the response body at 100 KB and then applies the selection expression. To send payloads larger than 100 KB, specify `$response.body` .
Context variable	$context.*variableName* or ${context.*variableName* }	The value of a supported context variable.
Stage variable	$stageVariables.*variableName* or ${stageVariables.*variableName* }	The value of a stage variable.
Static value	*string*	A constant value.

Note

To use multiple variables in a selection expression, enclose the variable in brackets. For example, `${request.path.name} ${request.path.id}`.

Reserved headers

The following headers are reserved. You can't configure request or response mappings for these headers.

- access-control-*
- apigw-*
- Authorization
- Connection
- Content-Encoding
- Content-Length
- Content-Location
- Forwarded
- Keep-Alive
- Origin
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailers
- Transfer-Encoding
- Upgrade
- x-amz-*
- x-amzn-*
- X-Forwarded-For
- X-Forwarded-Host
- X-Forwarded-Proto
- Via

Examples

The following AWS CLI examples configure parameter mappings. For example AWS CloudFormation templates, see GitHub.

Add a header to an API request

The following example adds a header named `header1` to an API request before it reaches your backend integration. API Gateway populates the header with the request ID.

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --request-parameters '{ "append:header.header1": "$context.requestId" }'
```

Rename a request header

The following example renames a request header from `header1` to `header2`.

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --request-parameters '{ "append:header.header2": "$request.header.header1",
 "remove:header.header1": "''"}'
```

Change the response from an integration

The following example configures response parameters for an integration. When the integrations returns a 500 status code, API Gateway changes the status code to 403, and adds `header11` to the response. When the integration returns a 404 status code, API Gateway adds an `error` header to the response.

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --response-parameters '{"500" : {"append:header.header1": "$context.requestId",
 "overwrite:statuscode" : "403"}, "404" : {"append:header.error" :
 "$stageVariables.environmentId"}  }'
```

Remove configured parameter mappings

The following example command removes previously configured request parameters for `append:header.header1`. It also removes previously configured response parameters for a 200 status code.

```
aws apigatewayv2 update-integration \
    --api-id abcdef123 \
    --integration-id hijk456 \
    --request-parameters '{"append:header.header1" : ""}' \
    --response-parameters '{"200" : {}}'
```

Working with OpenAPI definitions for HTTP APIs

You can define your HTTP API by using an OpenAPI 3.0 definition file. Then you can import the definition into API Gateway to create an API. To learn more about API Gateway extensions to OpenAPI, see *OpenAPI extensions*.

Importing an HTTP API

You can create an HTTP API by importing an OpenAPI 3.0 definition file.

To migrate from a REST API to an HTTP API, you can export your REST API as an OpenAPI 3.0 definition file. Then import the API definition as an HTTP API. To learn more about exporting a REST API, see Export a REST API from API Gateway.

> **Note**
>
> HTTP APIs support the same AWS variables as REST APIs. To learn more, see AWS variables for OpenAPI import.

Import validation information

As you import an API, API Gateway provides three categories of validation information.

Info

A property is valid according to the OpenAPI specification, but that property isn't supported for HTTP APIs.

For example, the following OpenAPI 3.0 snippet produces info on import because HTTP APIs don't support request validation. API Gateway ignores the `requestBody` and `schema` fields.

```
"paths": {
  "/": {
    "get": {
      "x-amazon-apigateway-integration": {
        "type": "AWS_PROXY",
        "httpMethod": "POST",
        "uri": "arn:aws:lambda:us-east-2:123456789012:function:HelloWorld",
        "payloadFormatVersion": "1.0"
      },
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Body"
            }
          }
        }
      }
    }
  }
  ...
},
"components": {
  "schemas": {
    "Body": {
      "type": "object",
      "properties": {
        "key": {
          "type": "string"
        }
      }
    }
    ...
  }
  ...
}
```

Warning

A property or structure is invalid according to the OpenAPI specification, but it doesn’t block API creation. You can specify whether API Gateway should ignore these warnings and continue creating the API, or stop creating the API on warnings.

The following OpenAPI 3.0 document produces warnings on import because HTTP APIs support only Lambda proxy and HTTP proxy integrations.

```
"x-amazon-apigateway-integration": {
  "type": "AWS",
  "httpMethod": "POST",
  "uri": "arn:aws:lambda:us-east-2:123456789012:function:HelloWorld",
  "payloadFormatVersion": "1.0"
}
```

Error

The OpenAPI specification is invalid or malformed. API Gateway can’t create any resources from the malformed document. You must fix the errors, and then try again.

The following API definition produces errors on import because HTTP APIs support only the OpenAPI 3.0 specification.

```
{
  "swagger": "2.0.0",
  "info": {
    "title": "My API",
    "description": "An Example OpenAPI definition for Errors/Warnings/ImportInfo",
    "version": "1.0"
  }
  ...
}
```

As another example, while OpenAPI allows users to define an API with multiple security requirements attached to a particular operation, API Gateway does not support this. Each operation can have only one of IAM authorization, a Lambda authorizer, or a JWT authorizer. Attempting to model multiple security requirements results in an error.

Import an API by using the AWS CLI

The following command imports the OpenAPI 3.0 definition file `api-definition.json` as an HTTP API.

Example

```
aws apigatewayv2 import-api --body file://api-definition.json
```

Example

You can import the following example OpenAPI 3.0 definition to create an HTTP API.

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Example Pet Store",
    "description": "A Pet Store API.",
    "version": "1.0"
  },
  "paths": {
    "/pets": {
      "get": {
        "operationId": "GET HTTP",
        "parameters": [
          {
            "name": "type",
            "in": "query",
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "page",
            "in": "query",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Pets"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "GET",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets",
          "payloadFormatVersion": 1.0
        }
      },
      "post": {
        "operationId": "Create Pet",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/NewPet"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/NewPetResponse"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "POST",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets",
          "payloadFormatVersion": 1.0
        }
      }
    },
    "/pets/{petId}": {
      "get": {
        "operationId": "Get Pet",
        "parameters": [
          {
            "name": "petId",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Pet"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "GET",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets/
{petId}",
          "payloadFormatVersion": 1.0
        }
      }
    }
  },
  "x-amazon-apigateway-cors": {
    "allowOrigins": [
      "*"
    ],
    "allowMethods": [
      "GET",
      "OPTIONS",
      "POST"
    ],
    "allowHeaders": [
      "x-amzm-header",
      "x-apigateway-header",
      "x-api-key",
      "authorization",
      "x-amz-date",
      "content-type"
    ]
  },
  "components": {
    "schemas": {
      "Pets": {
        "type": "array",
        "items": {
          "$ref": "#/components/schemas/Pet"
        }
      },
      "Empty": {
        "type": "object"
      },
      "NewPetResponse": {
        "type": "object",
        "properties": {
          "pet": {
            "$ref": "#/components/schemas/Pet"
          },
          "message": {
            "type": "string"
          }
        }
      },
      "Pet": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "type": {
            "type": "string"
          },
          "price": {
            "type": "number"
          }
        }
      },
      "NewPet": {
        "type": "object",
        "properties": {
          "type": {
            "$ref": "#/components/schemas/PetType"
          },
          "price": {
            "type": "number"
          }
        }
      },
      "PetType": {
        "type": "string",
        "enum": [
          "dog",
          "cat",
          "fish",
          "bird",
          "gecko"
        ]
      }
    }
  }
}
```

Exporting an HTTP API from API Gateway

After you've created an HTTP API, you can export an OpenAPI 3.0 definition of your API from API Gateway. You can either choose a stage to export, or export the latest configuration of your API. You can also import an exported API definition into API Gateway to create another, identical API. To learn more about importing API definitions, see Importing an HTTP API.

Export an OpenAPI 3.0 definition of a stage by using the AWS CLI

The following command exports an OpenAPI definition of an API stage named `prod` to a YAML file named `stage-definition.yaml`. The exported definition file includes API Gateway extensions by default.

```
aws apigatewayv2 export-api \
    --api-id api-id  \
    --output-type YAML  \
    --specification OAS30 \
    --stage-name prod \
    stage-definition.yaml
```

Export an OpenAPI 3.0 definition of your API's latest changes by using the AWS CLI

The following command exports an OpenAPI definition of an HTTP API to a JSON file named `latest-api-definition.json`. Because the command doesn't specify a stage, API Gateway exports the latest configuration of your API, whether it has been deployed to a stage or not. The exported definition file doesn't include API Gateway extensions.

```
aws apigatewayv2 export-api \
    --api-id api-id  \
    --output-type JSON  \
    --specification OAS30 \
    --no-include-extensions \
    latest-api-definition.json
```

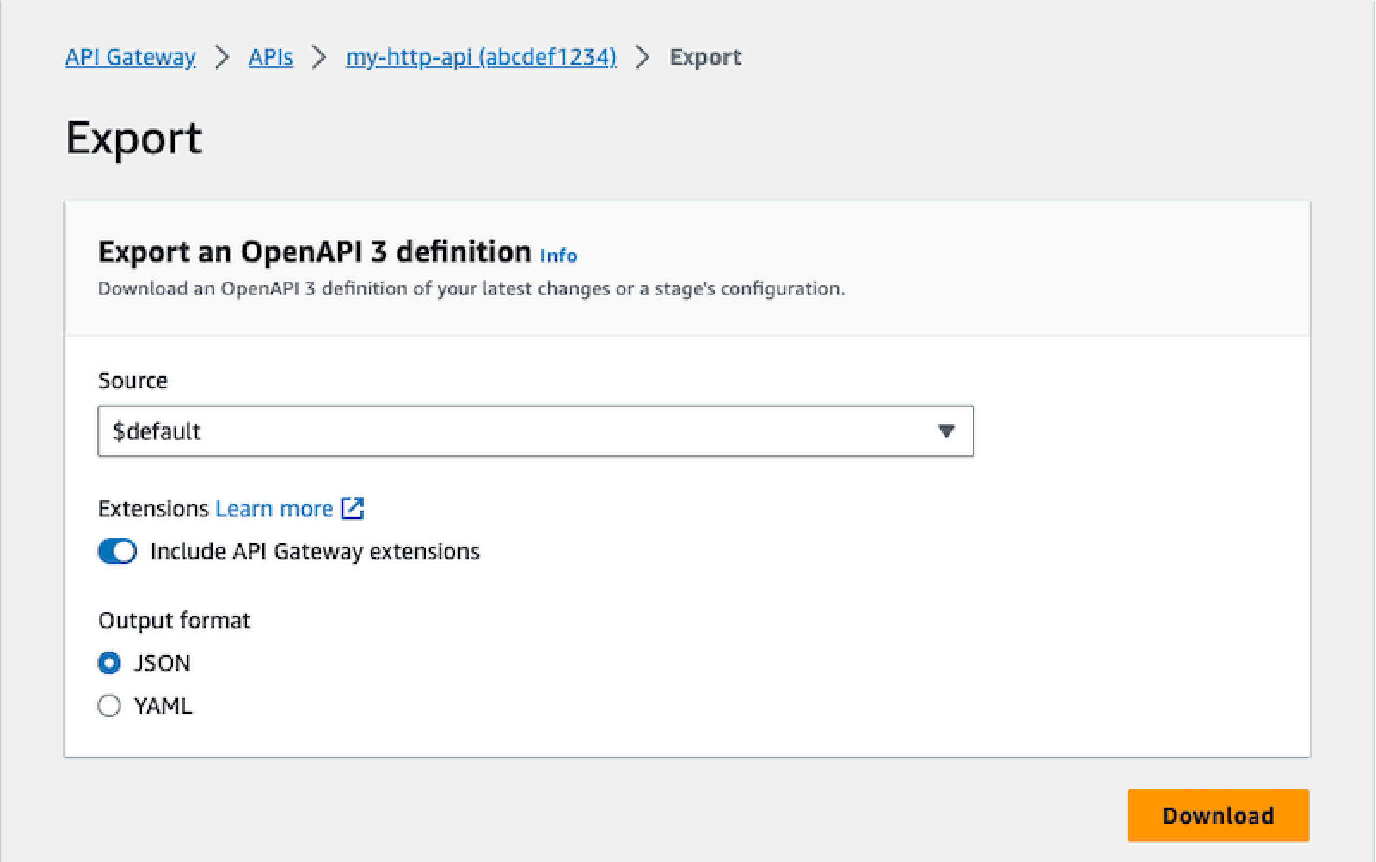
For more information, see ExportAPI in the *Amazon API Gateway Version 2 API Reference*.

Export an OpenAPI 3.0 definition by using the API Gateway console

The following procedure shows how to export an OpenAPI definition of an HTTP API.

To export an OpenAPI 3.0 definition using the API Gateway console

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose an HTTP API.
3. On the main navigation pane, under **Develop**, choose **Export**.
4. Select from the following options to export your API:



Export

Export an OpenAPI 3 definition Info

Download an OpenAPI 3 definition of your latest changes or a stage's configuration.

Source

$default ▼

Extensions Learn more

● Include API Gateway extensions

Output format

● JSON

○ YAML

   a. For **Source**, select a source for the OpenAPI 3.0 definition. You can choose a stage to export, or export the latest configuration of your API.
   b. Turn on **Include API Gateway extensions** to include API Gateway extensions.
   c. For **Output format**, select an output format.
5. Choose **Download**.

Publishing HTTP APIs for customers to invoke

You can use stages and custom domain names to publish your API for clients to invoke.

An API stage is a logical reference to a lifecycle state of your API (for example, `dev`, `prod`, `beta`, or `v2`). Each stage is a named reference to a deployment of the API and is made available for client applications to call. You can configure different integrations and settings for each stage of an API.

You can use custom domain names to provide a simpler, more intuitive URL for clients to invoke your API than the default URL, `https://api-id.execute-api.region.amazonaws.com/stage`.

> **Note**
>
> To augment the security of your API Gateway APIs, the `execute-api.{region}.amazonaws.com` domain is registered in the Public Suffix List (PSL). For further security, we recommend that you use cookies with a `__Host-` prefix if you ever need to set sensitive cookies in the default domain name for your API Gateway APIs. This practice will help to defend your domain against cross-site request forgery attempts (CSRF). For more information see the Set-Cookie page in the Mozilla Developer Network.

Topics

Working with stages for HTTP APIs

An API stage is a logical reference to a lifecycle state of your API (for example, `dev`, `prod`, `beta`, or `v2`). API stages are identified by their API ID and stage name, and they're included in the URL you use to invoke the API. Each stage is a named reference to a deployment of the API and is made available for client applications to call.

You can create a `$default` stage that is served from the base of your API's URL—for example, `https://{api_id}.execute-api.{region}.amazonaws.com/`. You use this URL to invoke an API stage.

A deployment is a snapshot of your API configuration. After you deploy an API to a stage, it’s available for clients to invoke. You must deploy an API for changes to take effect. If you enable automatic deployments, changes to an API are automatically released for you.

Stage variables

Stage variables are key-value pairs that you can define for a stage of an HTTP API. They act like environment variables and can be used in your API setup.

For example, you can define a stage variable, and then set its value as an HTTP endpoint for an HTTP proxy integration. Later, you can reference the endpoint by using the associated stage variable name. By doing this, you can use the same API setup with a different endpoint at each stage. Similarly, you can use stage variables to specify a different AWS Lambda function integration for each stage of your API.

> **Note**
>
> Stage variables are not intended to be used for sensitive data, such as credentials. To pass sensitive data to integrations, use an AWS Lambda authorizer. You can pass sensitive data to integrations in the output of the Lambda authorizer. To learn more, see the section called “Lambda authorizer response format”.

Examples

To use a stage variable to customize the HTTP integration endpoint, you must first set the name and value of the stage variable (for example, `url`) with a value of `example.com`. Next, set up an HTTP proxy integration. Instead of entering the endpoint's URL, you can tell API Gateway to use the stage variable value, **`http://${stageVariables.url}`**. This value tells API Gateway to substitute your stage variable `${}` at runtime, depending on the stage of your API.

You can reference stage variables in a similar way to specify a Lambda function name or an AWS role ARN.

When specifying a Lambda function name as a stage variable value, you must configure the permissions on the Lambda function manually. You can use the AWS Command Line Interface (AWS CLI) to do this.

```
aws lambda add-permission --function-name arn:aws:lambda:XXXXXX:your-lambda-function-
name --source-arn arn:aws:execute-api:us-east-1:YOUR_ACCOUNT_ID:api_id/*/HTTP_METHOD/
```

```
resource --principal apigateway.amazonaws.com --statement-id apigateway-access --action
 lambda:InvokeFunction
```

API Gateway stage variables reference

HTTP integration URIs

You can use a stage variable as part of an HTTP integration URI, as shown in the following examples.

- A full URI without protocol – `http://${stageVariables.<variable_name>}`
- A full domain – `http://${stageVariables.<variable_name>}/resource/operation`
- A subdomain – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- A path – `http://example.com/${stageVariables.<variable_name>}/bar`
- A query string – `http://example.com/foo?q=${stageVariables.<variable_name>}`

Lambda functions

You can use a stage variable in place of a Lambda function integration name or alias, as shown in the following examples.

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

Note

To use a stage variable for a Lambda function, the function must be in the same account as the API. Stage variables don't support cross-account Lambda functions.

AWS integration credentials

You can use a stage variable as part of an AWS user or role credential ARN, as shown in the following example.

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

Security policy for HTTP APIs

API Gateway enforces a security policy of `TLS_1_2` for all HTTP API endpoints.

A *security policy* is a predefined combination of minimum TLS version and cipher suites offered by Amazon API Gateway. The TLS protocol addresses network security problems such as tampering and eavesdropping between a client and server. When your clients establish a TLS handshake to your API through the custom domain, the security policy enforces the TLS version and cipher suite options your clients can choose to use. This security policy accepts TLS 1.2 and TLS 1.3 traffic and rejects TLS 1.0 traffic.

Supported TLS protocols and ciphers for HTTP APIs

The following table describes the supported TLS protocols and ciphers for HTTP APIs.

Security policy	TLS_1_2
TLS protocols	
TLSv1.3	♦
TLSv1.2	♦
TLS ciphers	
TLS-AES-128-GCM-SHA256	♦
TLS-AES-256-GCM-SHA384	♦
TLS-CHACHA20-POLY1305-SHA256	♦
ECDHE-ECDSA-AES128-GCM-SHA256	♦
ECDHE-RSA-AES128-GCM-SHA256	♦

Security policy	TLS_1_2
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256-GCM-SHA384	♦
ECDHE-RSA-AES256-GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL and RFC cipher names

OpenSSL and IETF RFC 5246 use different names for the same ciphers. For a list of the cipher names, see the section called “OpenSSL and RFC cipher names”.

Information about REST APIs and WebSocket APIs

For more information about REST APIs and WebSocket APIs, see the section called “Choosing a security policy” and the section called “Security policy for WebSocket APIs”.

Setting up custom domain names for HTTP APIs

Custom domain names are simpler and more intuitive URLs that you can provide to your API users.

After deploying your API, you (and your customers) can invoke the API using the default base URL of the following format:

```
https://api-id.execute-api.region.amazonaws.com/stage
```

where `api-id` is generated by API Gateway, `region` (AWS Region) is specified by you when creating the API, and `stage` is specified by you when deploying the API.

The hostname portion of the URL (that is, `api-id.execute-api.region.amazonaws.com`) refers to an API endpoint. The default API endpoint can be difficult to recall and not user-friendly.

With custom domain names, you can set up your API's hostname, and choose a base path (for example, `myservice`) to map the alternative URL to your API. For example, a more user-friendly API base URL can become:

```
https://api.example.com/myservice
```

> **Note**
>
> A custom domain can be associated with REST APIs and HTTP APIs. You can use API Gateway Version 2 APIs to create and manage Regional custom domain names for REST APIs and HTTP APIs.
> For HTTP APIs, TLS 1.2 is the only supported TLS version.

Register a domain name

You must have a registered internet domain name in order to set up custom domain names for your APIs. If needed, you can register an internet domain using Amazon Route 53 or using a third-party domain registrar of your choice. An API's custom domain name can be the name of a subdomain or the root domain (also known as "zone apex") of a registered internet domain.

After a custom domain name is created in API Gateway, you must create or update your DNS provider's resource record to map to your API endpoint. Without such a mapping, API requests bound for the custom domain name cannot reach API Gateway.

Regional custom domain names

When you create a custom domain name for a Regional API, API Gateway creates a Regional domain name for the API. You must set up a DNS record to map the custom domain name to the Regional domain name. You must also provide a certificate for the custom domain name.

Wildcard custom domain names

With wildcard custom domain names, you can support an almost infinite number of domain names without exceeding the default quota. For example, you could give each of your customers their own domain name, `customername.api.example.com`.

To create a wildcard custom domain name, specify a wildcard (*) as the first subdomain of a custom domain that represents all possible subdomains of a root domain.

For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`, which all route to the same domain.

Wildcard custom domain names support distinct configurations from API Gateway's standard custom domain names. For example, in a single AWS account, you can configure `*.example.com` and `a.example.com` to behave differently.

To create a wildcard custom domain name, you must provide a certificate issued by ACM that has been validated using either the DNS or the email validation method.

> **Note**
>
> You can't create a wildcard custom domain name if a different AWS account has created a custom domain name that conflicts with the wildcard custom domain name. For example, if account A has created `a.example.com`, then account B can't create the wildcard custom domain name `*.example.com`.
> If account A and account B share an owner, you can contact the AWS Support Center to request an exception.

Certificates for custom domain names

> **Important**
>
> You specify the certificate for your custom domain name. If your application uses certificate pinning, sometimes known as SSL pinning, to pin an ACM certificate, the application might not be able to connect to your domain after AWS renews the certificate. For more information, see Certificate pinning problems in the *AWS Certificate Manager User Guide*.

To provide a certificate for a custom domain name in a Region where ACM is supported, you must request a certificate from ACM. To provide a certificate for a Regional custom domain name in a Region where ACM is not supported, you must import a certificate to API Gateway in that Region.

To import an SSL/TLS certificate, you must provide the PEM-formatted SSL/TLS certificate body, its private key, and the certificate chain for the custom domain name. Each certificate stored in ACM is identified by its ARN. To use an AWS managed certificate for a domain name, you simply reference its ARN.

ACM makes it straightforward to set up and use a custom domain name for an API. You create a certificate for the given domain name (or import a certificate), set up the domain name in API Gateway with the ARN of the certificate provided by ACM, and map a base path under the custom domain name to a deployed stage of the API. With certificates issued by ACM, you do not have to worry about exposing any sensitive certificate details, such as the private key.

For details on setting up a custom domain name, see Getting certificates ready in AWS Certificate Manager and Setting up a regional custom domain name in API Gateway.

Working with API mappings for HTTP APIs

You use API mappings to connect API stages to a custom domain name. After you create a domain name and configure DNS records, you use API mappings to send traffic to your APIs through your custom domain name.

An API mapping specifies an API, a stage, and optionally a path to use for the mapping. For example, you can map the `production` stage of an API to `https://api.example.com/orders`.

You can map HTTP and REST API stages to the same custom domain name.

Before you create an API mapping, you must have an API, a stage, and a custom domain name. To learn more about creating a custom domain name, see the section called “Setting up a regional custom domain name”.

Routing API requests

You can configure API mappings with multiple levels, for example `orders/v1/items` and `orders/v2/items`.

For API mappings with multiple levels, API Gateway routes requests to the API mapping that has the longest matching path. API Gateway considers only the paths configured for API mappings, and not API routes, to select the API to invoke. If no path matches the request, API Gateway sends the request to the API that you've mapped to the empty path `(none)`.

For custom domain names that use API mappings with multiple levels, API Gateway routes requests to the API mapping that has the longest matching prefix.

For example, consider a custom domain name `https://api.example.com` with the following API mappings:

1. `(none)` mapped to API 1.
2. `orders` mapped to API 2.
3. `orders/v1/items` mapped to API 3.
4. `orders/v2/items` mapped to API 4.
5. `orders/v2/items/categories` mapped to API 5.

Request	Selected API	Explanation
`https://api.example.com/orders`	`API 2`	The request exactly matches this API mapping.
`https://api.example.com/orders/v1/items`	`API 3`	The request exactly matches this API mapping.
`https://api.example.com/orders/v2/items`	`API 4`	The request exactly matches this API mapping.
`https://api.example.com/orders/v1/items/123`	`API 3`	API Gateway chooses the mapping that has the longest matching path. The 123 at the end of the request doesn't affect the selection.
`https://api.example.com/orders/v2/items/categories/5`	`API 5`	API Gateway chooses the mapping that has the longest matching path.

Request	Selected API	Explanation
`https://api.example.com/customers`	`API 1`	API Gateway uses the empty mapping as a catch-all.
`https://api.example.com/ordersandmore`	`API 2`	API Gateway chooses the mapping that has the longest matching prefix. For a custom domain name configured with single-level mappings, such as only `https://api.example.com/orders` and `https://api.example.com/` , API Gateway would choose `API 1`, as there is no matching path with `ordersandmore` .

Restrictions

- In an API mapping, the custom domain name and mapped APIs must be in the same AWS account.
- API mappings must contain only letters, numbers, and the following characters: `$-_.+!*'()/`.
- The maximum length for the path in an API mapping is 300 characters.
- You can have 200 API mappings with multiple levels for each domain name.
- You can only map HTTP APIs to a regional custom domain name with the TLS 1.2 security policy.
- You can't map WebSocket APIs to the same custom domain name as an HTTP API or REST API.

Create an API mapping

To create an API mapping, you must first create a custom domain name, API, and stage. For information about creating a custom domain name, see the section called “Setting up a regional custom domain name”.

For example AWS Serverless Application Model templates that create all resources, see Sessions With SAM on GitHub.

AWS Management Console

To create an API mapping

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Custom domain names**.
3. Select a custom domain name that you've already created.
4. Choose **API mappings**.
5. Choose **Configure API mappings**.
6. Choose **Add new mapping**.
7. Enter an **API**, a **Stage**, and optionally a **Path**.
8. Choose **Save**.

AWS CLI

The following AWS CLI command creates an API mapping. In this example, API Gateway sends requests to `api.example.com/v1/orders` to the specified API and stage.

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1/orders \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

The following AWS CloudFormation example creates an API mapping.

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'orders/v2/items'
    ApiId: !Ref MyApi
    Stage: !Ref MyStage
```

Disabling the default endpoint for an HTTP API

By default, clients can invoke your API by using the `execute-api` endpoint that API Gateway generates for your API. To ensure that clients can access your API only by using a custom domain name, disable the default `execute-api` endpoint.

> **Note**
>
> When you disable the default endpoint, it affects all stages of an API.

The following AWS CLI command disables the default endpoint for an HTTP API.

```
aws apigatewayv2 update-api \
    --api-id abcdef123 \
    --disable-execute-api-endpoint
```

After you disable the default endpoint, you must deploy your API for the change to take effect, unless automatic deployments are enabled.

The following AWS CLI command creates a deployment.

```
aws apigatewayv2 create-deployment \
    --api-id abcdef123 \
    --stage-name dev
```

Protecting your HTTP API

API Gateway provides a number of ways to protect your API from certain threats, like malicious users or spikes in traffic. You can protect your API using strategies like setting throttling targets, and enabling mutual TLS. In this section you can learn how to enable these capabilities using API Gateway.

Topics

Throttling requests to your HTTP API

You can configure throttling for your APIs to help protect them from being overwhelmed by too many requests. Throttles are applied on a best-effort basis and should be thought of as targets rather than guaranteed request ceilings.

API Gateway throttles requests to your API using the token bucket algorithm, where a token counts for a request. Specifically, API Gateway examines the rate and a burst of request submissions against all APIs in your account, per Region. In the token bucket algorithm, a burst can allow predefined overrun of those limits, but other factors can also cause limits to be overrun in some cases.

When request submissions exceed the steady-state request rate and burst limits, API Gateway begins to throttle requests. Clients may receive `429 Too Many Requests` error responses at this point. Upon catching such exceptions, the client can resubmit the failed requests in a way that is rate limiting.

As an API developer, you can set the target limits for individual API stages or routes to improve overall performance across all APIs in your account.

Account-level throttling per Region

By default, API Gateway limits the steady-state requests per second (RPS) across all APIs within an AWS account, per Region. It also limits the burst (that is, the maximum bucket size) across all APIs within an AWS account, per Region. In API Gateway, the burst limit represents the target maximum number of concurrent request submissions that API Gateway will fulfill before returning `429 Too Many Requests` error responses. For more information on throttling quotas, see *Quotas and important notes*.

Per-account limits are applied to all APIs in an account in a specified Region. The account-level rate limit can be increased upon request - higher limits are possible with APIs that have shorter timeouts and smaller payloads. To request an increase of account-level throttling limits per Region, contact the AWS Support Center. For more information, see *Quotas and important notes*. Note that these limits can't be higher than the AWS throttling limits.

Route-level throttling

You can set route-level throttling to override the account-level request throttling limits for a specific stage or for individual routes in your API. The default route throttling limits can't exceed account-level rate limits.

You can configure route-level throttling by using the AWS CLI. The following command configures custom throttling for the specified stage and route of an API.

```
aws apigatewayv2 update-stage \
    --api-id a1b2c3d4 \
    --stage-name dev \
    --route-settings '{"GET /pets":
{"ThrottlingBurstLimit":100,"ThrottlingRateLimit":2000}}'
```

Configuring mutual TLS authentication for an HTTP API

Mutual TLS authentication requires two-way authentication between the client and the server. With mutual TLS, clients must present X.509 certificates to verify their identity to access your API. Mutual TLS is a common requirement for Internet of Things (IoT) and business-to-business applications.

You can use mutual TLS along with other authorization and authentication operations that API Gateway supports. API Gateway forwards the certificates that clients provide to Lambda authorizers and to backend integrations.

> ⚠ **Important**
>
> By default, clients can invoke your API by using the `execute-api` endpoint that API Gateway generates for your API. To ensure that clients can access your API only by using a custom domain name with mutual TLS, disable the default `execute-api` endpoint. To learn more, see the section called “Disable the default endpoint”.

Prerequisites for mutual TLS

To configure mutual TLS you need:

- A custom domain name
- At least one certificate configured in AWS Certificate Manager for your custom domain name
- A truststore configured and uploaded to Amazon S3

Custom domain names

To enable mutual TLS for a HTTP API, you must configure a custom domain name for your API. You can enable mutual TLS for a custom domain name, and then provide the custom domain name to clients. To access an API by using a custom domain name that has mutual TLS enabled, clients must present certificates that you trust in API requests. You can find more information at the section called "Custom domain names".

Using AWS Certificate Manager issued certificates

You can request a publicly trusted certificate directly from ACM or import public or self-signed certificates. To setup a certificate in ACM, go to ACM. If you would like to import a certificate, continue reading in the following section.

Using an imported or AWS Private Certificate Authority certificate

To use a certificate imported into ACM or a certificate from AWS Private Certificate Authority with mutual TLS, API Gateway needs an `ownershipVerificationCertificate` issued by ACM. This ownership certificate is only used to verify that you have permissions to use the domain name. It is not used for the TLS handshake. If you don't already have a `ownershipVerificationCertificate`, go to https://console.aws.amazon.com/acm/ to set one up.

You will need to keep this certificate valid for the lifetime of your domain name. If a certificate expires and auto-renew fails, all updates to the domain name will be locked. You will need to update the `ownershipVerificationCertificateArn` with a valid `ownershipVerificationCertificate` before you can make any other changes. The `ownershipVerificationCertificate` cannot be used as a server certificate for another mutual TLS domain in API Gateway. If a certificate is directly re-imported into ACM, the issuer must stay the same.

Configuring your truststore

Truststores are text files with a `.pem` file extension. They are a trusted list of certificates from Certificate Authorities. To use mutual TLS, create a truststore of X.509 certificates that you trust to access your API.

You must include the complete chain of trust, starting from the issuing CA certificate, up to the root CA certificate, in your truststore. API Gateway accepts client certificates issued by any CA present in the chain of trust. The certificates can be from public or private certificate authorities. Certificates can have a maximum chain length of four. You can also provide self-signed certificates. The following hashing algorithms are supported in the truststore:

- SHA-256 or stronger
- RSA-2048 or stronger

- ECDSA-256 or stronger

API Gateway validates a number of certificate properties. You can use Lambda authorizers to perform additional checks when a client invokes an API, including checking whether a certificate has been revoked. API Gateway validates the following properties:

Validation	Description
X.509 syntax	The certificate must meet X.509 syntax requirements.
Integrity	The certificate's content must not have been altered from that signed by the certificate authority from the truststore.
Validity	The certificate's validity period must be current.
Name chaining / key chaining	The names and subjects of certificates must form an unbroken chain. Certificates can have a maximum chain length of four.

Upload the truststore to an Amazon S3 bucket in a single file

Example certificates.pem

```
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
...
```

The following AWS CLI command uploads `certificates.pem` to your Amazon S3 bucket.

```
aws s3 cp certificates.pem s3://bucket-name
```

Configuring mutual TLS for a custom domain name

To configure mutual TLS for a HTTP API, you must use a Regional custom domain name for your API, with a minimum TLS version of 1.2. To learn more about creating and configuring a custom domain name, see the section called "Setting up a regional custom domain name".

> **Note**
>
> Mutual TLS isn't supported for private APIs.

After you've uploaded your truststore to Amazon S3, you can configure your custom domain name to use mutual TLS. Paste the following (slashes included) into a terminal:

```
aws apigatewayv2 create-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreUri=s3://bucket-name/key-name
```

After you create the domain name, you must configure DNS records and basepath mappings for API operations. To learn more, see Setting up a regional custom domain name in API Gateway.

Invoke an API by using a custom domain name that requires mutual TLS

To invoke an API with mutual TLS enabled, clients must present a trusted certificate in the API request. When a client attempts to invoke your API, API Gateway looks for the client certificate's issuer in your truststore. For API Gateway to proceed with the request, the certificate's issuer and the complete chain of trust up to the root CA certificate must be in your truststore.

The following example `curl` command sends a request to `api.example.com,` that includes `my-cert.pem` in the request. `my-key.key` is the private key for the certificate.

```
curl -v --key ./my-key.key --cert ./my-cert.pem api.example.com
```

Your API is invoked only if your truststore trusts the certificate. The following conditions will cause API Gateway to fail the TLS handshake and deny the request with a `403` status code. If your certificate:

- isn't trusted
- is expired
- doesn't use a supported algorithm

> **Note**
>
> API Gateway doesn't verify if a certificate has been revoked.

Updating your truststore

To update the certificates in your truststore, upload a new certificate bundle to Amazon S3. Then, you can update your custom domain name to use the updated certificate.

Use Amazon S3 versioning to maintain multiple versions of your truststore. When you update your custom domain name to use a new truststore version, API Gateway returns warnings if certificates are invalid.

API Gateway produces certificate warnings only when you update your domain name. API Gateway doesn’t notify you if a previously uploaded certificate expires.

The following AWS CLI command updates a custom domain name to use a new truststore version.

```
aws apigatewayv2 update-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreVersion='abcdef123'
```

Disable mutual TLS

To disable mutual TLS for a custom domain name, remove the truststore from your custom domain name, as shown in the following command.

```
aws apigatewayv2 update-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreUri=''
```

Troubleshooting certificate warnings

When creating a custom domain name with mutual TLS, API Gateway returns warnings if certificates in the truststore are not valid. This can also occur when updating a custom domain name to use a new truststore. The warnings indicate the issue with the certificate and the subject of the certificate that produced the warning. Mutual TLS is still enabled for your API, but some clients might not be able to access your API.

You'll need to decode the certificates in your truststore in order to identify which certificate produced the warning. You can use tools such as `openssl` to decode the certificates and identify their subjects.

The following command displays the contents of a certificate, including its subject:

```
openssl x509 -in certificate.crt -text -noout
```

Update or remove the certificates that produced warnings, and then upload a new truststore to Amazon S3. After uploading the new truststore, update your custom domain name to use the new truststore.

Troubleshooting domain name conflicts

The error "`The certificate subject <certSubject> conflicts with an existing certificate from a different issuer.`" means multiple Certificate Authorities have issued a certificate for this domain. For each subject in the certificate, there can only be one issuer in API Gateway for mutual TLS domains. You will need to get all of your certificates for that subject through a single issuer. If the problem is with a certificate you don't have control of but you can prove ownership of the domain name, contact AWS Support to open a ticket.

Troubleshooting domain name status messages

`PENDING_CERTIFICATE_REIMPORT`: This means you reimported a certificate to ACM and it failed validation because the new certificate has a SAN (subject alternative name) that is not covered by the `ownershipVerificationCertificate` or the subject or SANs in the certificate don't cover the domain name. Something might be configured incorrectly or an invalid certificate was imported. You need to reimport a valid certificate into ACM. For more information about validation see Validating domain ownership.

`PENDING_OWNERSHIP_VERIFICATION`: This means your previously verified certificate has expired and ACM was unable to auto-renew it. You will need to renew the certificate or request a new certificate. More information about certificate renewal can be found at ACM's troubleshooting managed certificate renewal guide.

Monitoring your HTTP API

You can use CloudWatch metrics and CloudWatch Logs to monitor HTTP APIs. By combining logs and metrics, you can log errors and monitor your API's performance.

> **Note**
>
> API Gateway might not generate logs and metrics in the following cases:
>
> - 413 Request Entity Too Large errors
> - Excessive 429 Too Many Requests errors
> - 400 series errors from requests sent to a custom domain that has no API mapping
> - 500 series errors caused by internal failures

Topics

Working with metrics for HTTP APIs

You can monitor API execution by using CloudWatch, which collects and processes raw data from API Gateway into readable, near-real-time metrics. These statistics are recorded for a period of 15 months so you can access historical information and gain a better perspective on how your web application or service is performing. By default, API Gateway metric data is automatically sent to CloudWatch in one-minute periods. To monitor your metrics, create a CloudWatch dashboard for your API. For more information about how to create a CloudWatch dashboard, see Creating a

CloudWatch dashboard in the *Amazon CloudWatch User Guide*. For more information, see What Is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

The following metrics are supported for HTTP APIs. You can also enable detailed metrics to write route-level metrics to Amazon CloudWatch.

Metric	Description
4xx	The number of client-side errors captured in a given period.
5xx	The number of server-side errors captured in a given period.
Count	The total number API requests in a given period.
IntegrationLatency	The time between when API Gateway relays a request to the backend and when it receives a response from the backend.
Latency	The time between when API Gateway receives a request from a client and when it returns a response to the client. The latency includes the integration latency and other API Gateway overhead.
DataProcessed	The amount of data processed in bytes.

You can use the dimensions in the following table to filter API Gateway metrics.

Dimension	Description
ApiId	Filters API Gateway metrics for an API with the specified API ID.
ApiId, Stage	Filters API Gateway metrics for an API stage with the specified API ID and stage ID.
ApiId, Method, Resource, Stage	Filters API Gateway metrics for an API method with the specified API ID, stage ID, resource path, and route ID. API Gateway will not send these metrics unless you have explicitly enabled detailed CloudWatch metrics. You can do this by calling the UpdateStage action of the API Gateway V2 REST API to update the `detailedMetricsEnabled` property to `true`. Alternatively, you can call the update-stage AWS CLI command to update the `DetailedMetricsEnabled` property to `true`. Enabling such metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing.

Configuring logging for an HTTP API

You can turn on logging to write logs to CloudWatch Logs. You can use logging variables to customize the content of your logs.

To turn on logging for an HTTP API, you must do the following.

1. Ensure that your user has the required permissions to activate logging.
2. Create a CloudWatch Logs log group.
3. Provide the ARN of the CloudWatch Logs log group for a stage of your API.

Permissions to activate logging

To turn on logging for an API, your user must have the following permissions.

Example

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:GetLogEvents",
                "logs:FilterLogEvents"
            ],
            "Resource": "arn:aws:logs:us-east-2:123456789012:log-group:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:PutResourcePolicy",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:CreateLogGroup",
                "logs:DescribeResourcePolicies",
                "logs:GetLogDelivery",
                "logs:ListLogDeliveries"
            ],
            "Resource": "*"
        }
    ]
}
```

Create a log group and activate logging for HTTP APIs

You can create a log group and activate access logging using the AWS Management Console or the AWS CLI.

AWS Management Console

1. Create a log group.

   To learn how to create a log group using the console, see Create a Log Group in Amazon CloudWatch Logs User Guide.
2. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
3. Choose an HTTP API.
4. Under the **Monitor** tab in the primary navigation panel, choose **Logging**.
5. Select a stage to activate logging and choose **Select**.
6. Choose **Edit** to activate access logging.
7. Turn on **Access logging**, enter a CloudWatch Logs, and select a log format.
8. Choose **Save**.

AWS CLI

The following AWS CLI command creates a log group.

```
aws logs create-log-group --log-group-name my-log-group
```

You need the Amazon Resource Name (ARN) for your log group to turn on logging. The ARN format is arn:aws:logs:*region*:*account-id*:log-group:*log-group-name*.

The following AWS CLI command turns on logging for the `$default` stage of an HTTP API.

```
aws apigatewayv2 update-stage --api-id abcdef \
    --stage-name '$default' \
    --access-log-settings '{"DestinationArn": "arn:aws:logs:region:account-
id:log-group:log-group-name", "Format": "$context.identity.sourceIp - -
 [$context.requestTime] \"$context.httpMethod $context.routeKey $context.protocol\"
 $context.status $context.responseLength $context.requestId"}'
```

Example log formats

Examples of some common access log formats are available in the API Gateway console and are listed as follows.

- CLF (Common Log Format):

```
$context.identity.sourceIp - - [$context.requestTime] "$context.httpMethod
 $context.routeKey $context.protocol" $context.status $context.responseLength
 $context.requestId $context.extendedRequestId
```

- JSON:

```
{ "requestId":"$context.requestId", "ip": "$context.identity.sourceIp",
 "requestTime":"$context.requestTime",
 "httpMethod":"$context.httpMethod","routeKey":"$context.routeKey",
 "status":"$context.status","protocol":"$context.protocol",
 "responseLength":"$context.responseLength", "extendedRequestId":
 "$context.extendedRequestId" }
```

- XML:

```
<request id="$context.requestId"> <ip>$context.identity.sourceIp</ip> <requestTime>
$context.requestTime</requestTime> <httpMethod>$context.httpMethod</httpMethod>
 <routeKey>$context.routeKey</routeKey> <status>$context.status</status> <protocol>
$context.protocol</protocol> <responseLength>$context.responseLength</responseLength>
 <extendedRequestId>$context.extendedRequestId</extendedRequestId> </request>
```

- CSV (comma-separated values):

```
$context.identity.sourceIp,$context.requestTime,$context.httpMethod,
$context.routeKey,$context.protocol,$context.status,$context.responseLength,
$context.requestId,$context.extendedRequestId
```

Customizing HTTP API access logs

You can use the following variables to customize HTTP API access logs. To learn more about access logs for HTTP APIs, see Configuring logging for an HTTP API.

Parameter	Description
`$context.accountId`	The API owner's AWS account ID.
`$context.apiId`	The identifier API Gateway assigns to your API.
`$context.authorizer.claims.` *`property`*	A property of the claims returned from the JSON Web Token (JWT) after the method caller is successfully authenticated, such as `$context.authorizer.claims.username` . For more information, see Controlling access to HTTP APIs with JWT authorizers. **Note** Calling `$context.authorizer.claims` returns null.
`$context.authorizer.error`	The error message returned from an authorizer.
`$context.authorizer.principalId`	The principal user identification that a Lambda authorizer returns.

`$context.authorizer.` *`property`*	The value of the specified key-value pair of the `context` map returned from an API Gateway Lambda authorizer function. For example, if the authorizer returns the following `context` map:
	`"context" : {`
	`  "key": "value",`
	`  "numKey": 1,`
	`  "boolKey": true`

Parameter	Description
	`}` calling `$context.authorizer.key` returns the `"value"` string, calling `$context.authorizer.numKey` returns the 1, and calling `$context.authorizer.boolKey` returns `true`.
`$context.awsEndpointRequestId`	The AWS endpoint's request ID from the `x-amz-request-id` or `x-amzn-requestId` header.
`$context.awsEndpointRequestId2`	The AWS endpoint's request ID from the `x-amz-id-2` header.
`$context.customDomain.basePathMatched`	The path for an API mapping that an incoming request matched. Applicable when a client uses a custom domain name to access an API. For example if a client sends a request to `https://api.example.com/v1/orders/1234`, and the request matches the API mapping with the path `v1/orders`, the value is `v1/orders`. To learn more, see the section called "API mappings".
`$context.dataProcessed`	The amount of data processed in bytes.
`$context.domainName`	The full domain name used to invoke the API. This should be the same as the incoming `Host` header.
`$context.domainPrefix`	The first label of the `$context.domainName`.
`$context.error.message`	A string that contains an API Gateway error message.

Parameter	Description
`$context.error.messageString`	The quoted value of `$context.error.message` , namely `"$context.error.message"` .
`$context.error.responseType`	A type of `GatewayResponse` . For more information, see the section called "Metrics" and the section called "Setting up gateway responses to customize error responses".
`$context.extendedRequestId`	Equivalent to `$context.requestId` .
`$context.httpMethod`	The HTTP method used. Valid values include: `DELETE`, `GET`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, and PUT.
`$context.identity.accountId`	The AWS account ID associated with the request. Supported for routes that use IAM authorization.
`$context.identity.caller`	The principal identifier of the caller that signed the request. Supported for routes that use IAM authorization.

Parameter	Description
`$context.identity.cognitoAuthenticationProvider`	A comma-separated list of the Amazon Cognito authentication providers used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For example, for an identity from an Amazon Cognito user pool, `cognito-idp. *region*.amazonaws.com/ *user_pool_id* ,cognito-idp. *region*.amazonaws.com/ *user_pool_id* :CognitoSignIn: *token subject claim*` For information, see Using Federated Identities in the *Amazon Cognito Developer Guide*.
`$context.identity.cognitoAuthenticationType`	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials. Possible values include `authenticated` for authenticated identities and `unauthenticated` for unauthenticated identities.
`$context.identity.cognitoIdentityId`	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.cognitoIdentityPoolId`	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.principalOrgId`	The AWS organization ID. Supported for routes that use IAM authorization.

Parameter	Description
`$context.identity.clientCert.clientCertPem`	The PEM-encoded client certificate that the client presented during mutual TLS authentication. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.clientCert.subjectDN`	The distinguished name of the subject of the certificate that a client presents. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.clientCert.issuerDN`	The distinguished name of the issuer of the certificate that a client presents. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.clientCert.serialNumber`	The serial number of the certificate. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.clientCert.validity.notBefore`	The date before which the certificate is invalid. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.clientCert.validity.notAfter`	The date after which the certificate is invalid. Present when a client accesses an API by using a custom domain name that has mutual TLS enabled.
`$context.identity.sourceIp`	The source IP address of the immediate TCP connection making the request to API Gateway endpoint.

Parameter	Description
`$context.identity.user`	The principal identifier of the user that will be authorized against resource access. Supported for routes that use IAM authorization.
`$context.identity.userAgent`	The [`User-Agent`](#) header of the API caller.
`$context.identity.userArn`	The Amazon Resource Name (ARN) of the effective user identified after authentication. Supported for routes that use IAM authorization. For more information, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html.
`$context.integration.error`	The error message returned from an integration. Equivalent to `$context.integrationErrorMessage` .
`$context.integration.integrationStatus`	For Lambda proxy integration, the status code returned from AWS Lambda, not from the backend Lambda function code.
`$context.integration.latency`	The integration latency in ms. Equivalent to `$context.integrationLatency` .
`$context.integration.requestId`	The AWS endpoint's request ID. Equivalent to `$context.awsEndpointRequestId` .
`$context.integration.status`	The status code returned from an integration. For Lambda proxy integrations, this is the status code that your Lambda function code returns.
`$context.integrationErrorMessage`	A string that contains an integration error message.
`$context.integrationLatency`	The integration latency in ms.

Parameter	Description
`$context.integrationStatus`	For Lambda proxy integration, this parameter represents the status code returned from AWS Lambda, not from the backend Lambda function.
`$context.path`	The request path. For example, `/{stage}/root/child` .
`$context.protocol`	The request protocol, for example, `HTTP/1.1`. **Note** API Gateway APIs can accept HTTP/2 requests, but API Gateway sends requests to backend integrations using HTTP/1.1. As a result, the request protocol is logged as HTTP/1.1 even if a client sends a request that uses HTTP/2.
`$context.requestId`	The ID that API Gateway assigns to the API request.
`$context.requestTime`	The CLF-formatted request time (`dd/MMM/yyyy:HH:mm:ss +-hhmm`).
`$context.requestTimeEpoch`	The Epoch-formatted request time.
`$context.responseLatency`	The response latency in ms.
`$context.responseLength`	The response payload length in bytes.
`$context.routeKey`	The route key of the API request, for example `/pets`.

Parameter	Description
`$context.stage`	The deployment stage of the API request (for example, `beta` or `prod`).
`$context.status`	The method response status.

Troubleshooting issues with HTTP APIs

The following topics provide troubleshooting advice for errors and issues that you might encounter when using HTTP APIs.

Topics

- Troubleshooting issues with HTTP API Lambda integrations
- Troubleshooting issues with HTTP API JWT authorizers

Troubleshooting issues with HTTP API Lambda integrations

The following provides troubleshooting advice for errors and issues that you might encounter when using AWS Lambda integrations with HTTP APIs.

Issue: My API with a Lambda integration returns `{"message":"Internal Server Error"}`

To troubleshoot the internal server error, add the `$context.integrationErrorMessage` logging variable to your log format, and view your HTTP API's logs. To achieve this, do the following:

To create a log group by using the AWS Management Console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Log groups**.
3. Choose **Create log group**.
4. Enter a log group name, and then choose **Create**.

5. Note the Amazon Resource Name (ARN) for your log group. The ARN format is arn:aws:logs:*region*: *account-id*:log-group:*log-group-name*. You need the log group ARN to enable access logging for your HTTP API.

To add the `$context.integrationErrorMessage` logging variable

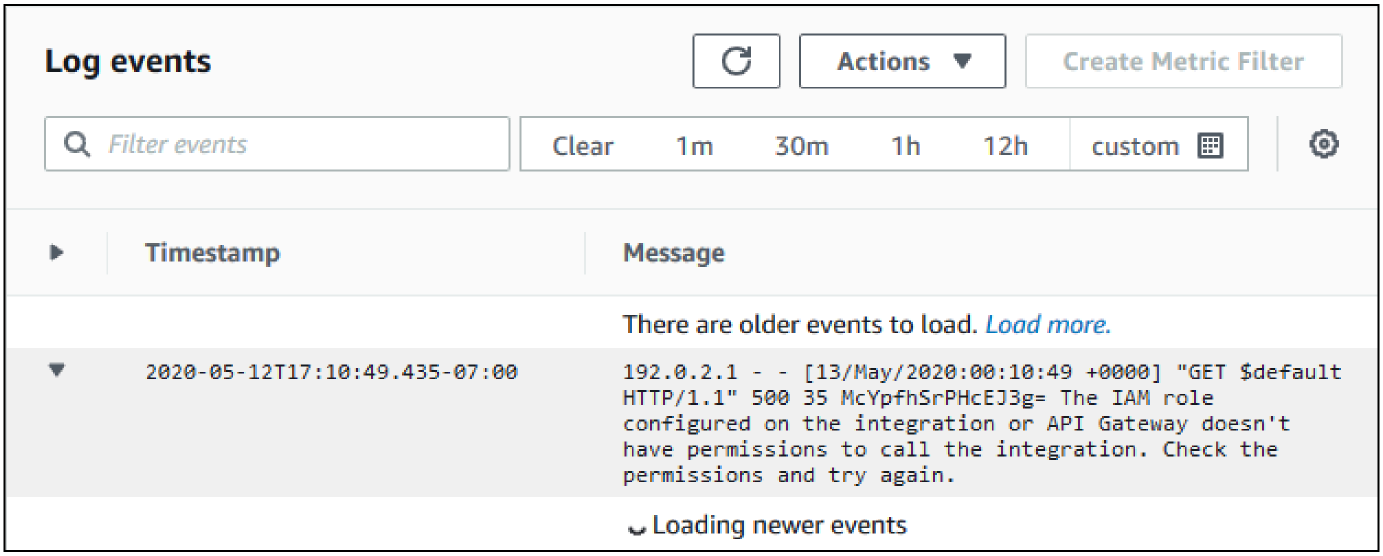
1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your HTTP API.
3. Under **Monitor**, choose **Logging**.
4. Select a stage of your API.
5. Choose **Edit**, and then enable access logging.
6. For **Log destination**, enter the ARN of the log group that you created in the previous step.
7. For **Log format**, choose **CLF**. API Gateway creates an example log format.
8. Add `$context.integrationErrorMessage` to the end of the log format.
9. Choose **Save**.

To view your API's logs

1. Generate logs. Use a browser or `curl` to invoke your API.

   ```
   $curl https://api-id.execute-api.us-west-2.amazonaws.com/route
   ```

2. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
3. Choose your HTTP API.
4. Under **Monitor**, choose **Logging**.
5. Select the stage of your API for which you enabled logging.
6. Choose **View logs in CloudWatch**.
7. Choose the latest log stream to view your HTTP API's logs.
8. Your log entry should look similar to the following:



```
2020-05-12T17:10:49.435-07:00    192.0.2.1 - - [13/May/2020:00:10:49 +0000] "GET $default
                                 HTTP/1.1" 500 35 McYpfhSrPHcEJ3g= The IAM role
                                 configured on the integration or API Gateway doesn't
                                 have permissions to call the integration. Check the
                                 permissions and try again.
```

Because we added `$context.integrationErrorMessage` to the log format, we see an error message in our logs that summarizes the problem.

Your logs might include a different error message that indicates that there's a problem with your Lambda function code. In that case, check your Lambda function code, and verify that your Lambda function returns a response in the required format. If your logs don't include an error message, add `$context.error.message` and `$context.error.responseType` to your log format for more information to help troubleshoot.

In this case, the logs show that API Gateway didn't have the required permissions to invoke the Lambda function.

When you create a Lambda integration in the API Gateway console, API Gateway automatically configures permissions to invoke the Lambda function. When you create a Lambda integration by using the AWS CLI, AWS CloudFormation, or an SDK, you must grant permissions for API Gateway to invoke the function. The following example AWS CLI commands grant permission for different HTTP API routes to invoke a Lambda function.

Example Example – For the `$default` stage and `$default` route of an HTTP API

```
aws lambda add-permission \
    --function-name my-function \
    --statement-id apigateway-invoke-permissions \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-id/\$default/\
$default"
```

Example Example – For the `prod` stage and `test` route of an HTTP API

```
aws lambda add-permission \
    --function-name my-function \
    --statement-id apigateway-invoke-permissions \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-id/prod/*/test"
```

Confirm the function policy in the **Permissions** tab of the Lambda console.

Try invoking your API again. You should see your Lambda function's response.

Troubleshooting issues with HTTP API JWT authorizers

The following provides troubleshooting advice for errors and issues that you might encounter when using JSON Web Token (JWT) authorizers with HTTP APIs.

Issue: My API returns 401 `{"message":"Unauthorized"}`

Check the `www-authenticate` header in the response from the API.

The following command uses `curl` to send a request to an API with a JWT authorizer that uses `$request.header.Authorization` as its identity source.

```
$curl -v -H "Authorization: token" https://api-id.execute-api.us-
west-2.amazonaws.com/route
```

The response from the API includes a `www-authenticate` header.

```
...
< HTTP/1.1 401 Unauthorized
< Date: Wed, 13 May 2020 04:07:30 GMT
< Content-Length: 26
< Connection: keep-alive
< www-authenticate: Bearer scope="" error="invalid_token" error_description="the token
 does not have a valid audience"
< apigw-requestid: Mc7UVioPPHcEKPA=
<
* Connection #0 to host api-id.execute-api.us-west-2.amazonaws.com left intact
{"message":"Unauthorized"}}
```

In this case, the `www-authenticate` header shows that the token wasn't issued for a valid audience. For API Gateway to authorize a request, the JWT's `aud` or `client_id` claim must match one of the audience entries that's configured for the authorizer. API Gateway validates `client_id` only if `aud` is not present. When both `aud` and `client_id` are present, API Gateway evaluates `aud`.

You can also decode a JWT and verify that it matches the issuer, audience, and scopes that your API requires. The website jwt.io can debug JWTs in the browser. The OpenID Foundation maintains a list of libraries for working with JWTs.

To learn more about JWT authorizers, see Controlling access to HTTP APIs with JWT authorizers.

Working with WebSocket APIs

A WebSocket API in API Gateway is a collection of WebSocket routes that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. You can use API Gateway features to help you with all aspects of the API lifecycle, from creation through monitoring your production APIs.

API Gateway WebSocket APIs are bidirectional. A client can send messages to a service, and services can independently send messages to clients. This bidirectional behavior enables richer client/service interactions because services can push data to clients without requiring clients to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms.

For an example app to get started with, see Tutorial: Building a serverless chat app with a WebSocket API, Lambda and DynamoDB.

In this section, you can learn how to develop, publish, protect, and monitor your WebSocket APIs using API Gateway.

Topics

About WebSocket APIs in API Gateway

In API Gateway you can create a WebSocket API as a stateful frontend for an AWS service (such as Lambda or DynamoDB) or for an HTTP endpoint. The WebSocket API invokes your backend based on the content of the messages it receives from client apps.

Unlike a REST API, which receives and responds to requests, a WebSocket API supports two-way communication between client apps and your backend. The backend can send callback messages to connected clients.

In your WebSocket API, incoming JSON messages are directed to backend integrations based on routes that you configure. (Non-JSON messages are directed to a `$default` route that you configure.)

A *route* includes a *route key*, which is the value that is expected once a *route selection expression* is evaluated. The `routeSelectionExpression` is an attribute defined at the API level. It specifies a JSON property that is expected to be present in the message payload. For more information about route selection expressions, see the section called “”.

For example, if your JSON messages contain an `action` property, and you want to perform different actions based on this property, your route selection expression might be `${request.body.action}`. Your routing table would specify which action to perform by matching the value of the `action` property against the custom route key values that you have defined in the table.

There are three predefined routes that can be used: `$connect`, `$disconnect`, and `$default`. In addition, you can create custom routes.

- API Gateway calls the `$connect` route when a persistent connection between the client and a WebSocket API is being initiated.
- API Gateway calls the `$disconnect` route when the client or the server disconnects from the API.
- API Gateway calls a custom route after the route selection expression is evaluated against the message if a matching route is found; the match determines which integration is invoked.
- API Gateway calls the `$default` route if the route selection expression cannot be evaluated against the message or if no matching route is found.

For more information about the `$connect` and `$disconnect` routes, see the section called “Managing connected users and client apps”.

For more information about the `$default` route and custom routes, see the section called “Invoking your backend integration”.

Backend services can send data to connected client apps. For more information, see the section called “Sending data from backend services to connected clients”.

Managing connected users and client apps: `$connect` and `$disconnect` routes

Topics

The `$connect` route

Client apps connect to your WebSocket API by sending a WebSocket upgrade request. If the request succeeds, the `$connect` route is executed while the connection is being established.

Because the WebSocket connection is a stateful connection, you can configure authorization on the `$connect` route only. `AuthN`/`AuthZ` will be performed only at connection time.

Until execution of the integration associated with the `$connect` route is completed, the upgrade request is pending and the actual connection will not be established. If the `$connect` request fails (e.g., due to `AuthN`/`AuthZ` failure or an integration failure), the connection will not be made.

> **Note**
>
> If authorization fails on `$connect`, the connection will not be established, and the client will receive a `401` or `403` response.

Setting up an integration for `$connect` is optional. You should consider setting up a `$connect` integration if:

- You want to enable clients to specify subprotocols by using the `Sec-WebSocket-Protocol` field. For example code, see Setting up a `$connect` route that requires a WebSocket subprotocol.
- You want to be notified when clients connect.
- You want to throttle connections or control who connects.
- You want your backend to send messages back to clients using a callback URL.

- You want to store each connection ID and other information into a database (for example, Amazon DynamoDB).

Passing connection information from the `$connect` route

You can use both proxy and non-proxy integrations to pass information from the `$connect` route to a database or other AWS service.

To pass connection information using a proxy integration

You can access the connection information from a Lambda proxy integration in the event. Use another AWS service or AWS Lambda function to post to the connection.

The following Lambda function shows how to use the `requestContext` object to log the connection ID, domain name, stage name, and query strings.

Node.js

```
 export const handler = async(event, context) => {
    const connectId = event["requestContext"]["connectionId"]
    const domainName = event["requestContext"]["domainName"]
    const stageName = event["requestContext"]["stage"]
    const qs = event['queryStringParameters']
    console.log('Connection ID: ', connectId, 'Domain Name: ', domainName, 'Stage
 Name: ', stageName, 'Query Strings: ', qs )
    return {"statusCode" : 200}
};
```

Python

```
import json
import logging
logger = logging.getLogger()
logger.setLevel("INFO")


def lambda_handler(event, context):
    connectId = event["requestContext"]["connectionId"]
    domainName = event["requestContext"]["domainName"]
    stageName = event["requestContext"]["stage"]
    qs = event['queryStringParameters']
    connectionInfo = {
        'Connection ID': connectId,
        'Domain Name': domainName,
        'Stage Name': stageName,
        'Query Strings': qs}
    logging.info(connectionInfo)
    return {"statusCode": 200}
```

To pass connection information using a non-proxy integration

- You can access the connection information with a non-proxy integration. Set up the integration request and provide a WebSocket API request template. The following Velocity Template Language (VTL) mapping template provides an integration request. This request sends the following details to a non-proxy integration:

  - Connection ID
  - Domain name
  - Stage name
  - Path
  - Headers
  - Query strings

  This request sends the connection ID, domain name, stage name, paths, headers, and query strings to a non-proxy integration.

  ```
  {
      "connectionId": "$context.connectionId",
      "domain": "$context.domainName",
      "stage": "$context.stage",
      "params": "$input.params()"
  }
  ```

  For more information about setting up data transformations, see the section called “Data transformations”.

  To complete the integration request, set `StatusCode: 200` for the integration response. To learn more about setting up an integration response, see Set up an integration response using the API Gateway console.

The $disconnect route

The $disconnect route is executed after the connection is closed.

The connection can be closed by the server or by the client. As the connection is already closed when it is executed, $disconnect is a best-effort event. API Gateway will try its best to deliver the $disconnect event to your integration, but it cannot guarantee delivery.

The backend can initiate disconnection by using the @connections API. For more information, see the section called “Use @connections commands in your backend service”.

Invoking your backend integration: $default Route and custom routes

Topics

- Using routes to process messages
- The $default route
- Custom routes
- Using API Gateway WebSocket API integrations to connect to your business logic
- Important differences between WebSocket APIs and REST APIs

Using routes to process messages

In API Gateway WebSocket APIs, messages can be sent from the client to your backend service and vice versa. Unlike HTTP's request/response model, in WebSocket the backend can send messages to the client without the client taking any action.

Messages can be JSON or non-JSON. However, only JSON messages can be routed to specific integrations based on message content. Non-JSON messages are passed through to the backend by the $default route.

> **Note**
>
> API Gateway supports message payloads up to 128 KB with a maximum frame size of 32 KB. If a message exceeds 32 KB, you must split it into multiple frames, each 32 KB or smaller. If a larger message (or frame) is received, the connection is closed with code 1009.

Currently binary payloads are not supported. If a binary frame is received, the connection is closed with code 1003. However, it is possible to convert binary payloads to text. See the section called “Binary media types”.

With WebSocket APIs in API Gateway, JSON messages can be routed to execute a specific backend service based on message content. When a client sends a message over its WebSocket connection, this results in a *route request* to the WebSocket API. The request will be matched to the route with the corresponding route key in API Gateway. You can set up a route request for a WebSocket API in the API Gateway console, by using the AWS CLI, or by using an AWS SDK.

Note

In the AWS CLI and AWS SDKs, you can create routes before or after you create integrations. Currently the console does not support reuse of integrations, so you must create the route first and then create the integration for that route.

You can configure API Gateway to perform validation on a route request before proceeding with the integration request. If the validation fails, API Gateway fails the request without calling your backend, sends a `"Bad request body"` gateway response similar to the following to the client, and publishes the validation results in CloudWatch Logs:

```
{"message" : "Bad request body", "connectionId": "{connectionId}", "messageId":
 "{messageId}"}
```

This reduces unnecessary calls to your backend and lets you focus on the other requirements of your API.

You can also define a route response for your API's routes to enable two-way communication. A route response describes what data will be sent to your client upon completion of a particular route's integration. It is not necessary to define a response for a route if, for example, you want a client to send messages to your backend without receiving a response (one-way communication). However, if you don't provide a route response, API Gateway won't send any information about the result of your integration to your clients.

The `$default` route

Every API Gateway WebSocket API can have a `$default` route. This is a special routing value that can be used in the following ways:

- You can use it together with defined route keys, to specify a "fallback" route (for example, a generic mock integration that returns a particular error message) for incoming messages that don't match any of the defined route keys.
- You can use it without any defined route keys, to specify a proxy model that delegates routing to a backend component.
- You can use it to specify a route for non-JSON payloads.

Custom routes

If you want to invoke a specific integration based on message content, you can do so by creating a custom route.

A custom route uses a route key and integration that you specify. When an incoming message contains a JSON property, and that property evaluates to a value that matches the route key value, API Gateway invokes the integration. (For more information, see the section called "About WebSocket APIs".)

For example, suppose you wanted to create a chat room application. You might start by creating a WebSocket API whose route selection expression is `$request.body.action`. You could then define two routes: `joinroom` and `sendmessage`. A client app might invoke the `joinroom` route by sending a message such as the following:

```
{"action":"joinroom","roomname":"developers"}
```

And it might invoke the `sendmessage` route by sending a message such as the following:

```
{"action":"sendmessage","message":"Hello everyone"}
```

Using API Gateway WebSocket API integrations to connect to your business logic

After setting up a route for an API Gateway WebSocket API, you must specify the integration you'd like to use. As with a route, which can have a route request and a route response, an integration can have an *integration request* and an *integration response*. An *integration request* contains the information expected by your backend in order to process the request that came from your client. An *integration response* contains the data that your backend returns to API Gateway, and that may be used to construct a message to send to the client (if a route response is defined).

For more information about setting up integrations, see the section called "Integrations".

Important differences between WebSocket APIs and REST APIs

Integrations for WebSocket APIs are similar to integrations for REST APIs, except for the following differences:

- Currently, in the API Gateway console you must create a route first and then create an integration as that route's target. However, in the API and CLI, you can create routes and integrations independently, in any order.
- You can use a single integration for multiple routes. For example, if you have a set of actions that closely relate to each other, you might want all of those routes to go to a single Lambda function. Rather than defining the details of the integration multiple times, you can specify it once and assign it to each of the related routes.

> **Note**
>
> Currently the console does not support reuse of integrations, so you must create the route first and then create the integration for that route.
> In the AWS CLI and AWS SDKs, you can reuse an integration by setting the route's target to a value of `"integrations/{integration-id}"`, where `{integration-id}"` is the unique ID of the integration to be associated with the route.

- API Gateway provides multiple selection expressions you can use in your routes and integrations. You don't need to rely on the content type to select an input template or output mapping. As with route selection expressions, you can define a selection expression to be evaluated by API Gateway to choose the right item. All of them will fall back to the `$default` template if a matching template is not found.
  - In integration requests, the template selection expression supports `$request.body.<json_path_expression>` and static values.
  - In integration responses, the template selection expression supports `$request.body.<json_path_expression>`, `$integration.response.statuscode`, `$integration.response.header.<headerName>`, and static values.

In the HTTP protocol, in which requests and responses are sent synchronously; communication is essentially one-way. In the WebSocket protocol, communication is two-way. Responses are asynchronous and are not necessarily received by the client in the same order as the client's messages were sent. In addition, the backend can send messages to the client.

> **Note**
>
> For a route that is configured to use `AWS_PROXY` or `LAMBDA_PROXY` integration, communication is one-way, and API Gateway will not pass the backend response through to the route response automatically. For example, in the case of `LAMBDA_PROXY` integration, the body that the Lambda function returns will not be returned to the client. If you want the client to receive integration responses, you must define a route response to make two-way communication possible.

Sending data from backend services to connected clients

API Gateway WebSocket APIs offer the following ways for you to send data from backend services to connected clients:

- An integration can send a response, which is returned to the client by a route response that you have defined.
- You can use the `@connections` API to send a POST request. For more information, see the section called "Use `@connections` commands in your backend service".

WebSocket selection expressions in API Gateway

Topics

API Gateway uses selection expressions as a way to evaluate request and response context and produce a key. The key is then used to select from a set of possible values, typically provided by you, the API developer. The exact set of supported variables will vary depending on the particular expression. Each expression is discussed in more detail below.

For all of the expressions, the language follows the same set of rules:

- A variable is prefixed with `"$"`.
- Curly braces can be used to explicitly define variable boundaries, e.g., `"${request.body.version}-beta"`.
- Multiple variables are supported, but evaluation occurs only once (no recursive evaluation).
- A dollar sign ($) can be escaped with `"\"`. This is most useful when defining an expression that maps to the reserved `$default` key, e.g., `"\$default"`.
- In some cases, a pattern format is required. In this case, the expression should be wrapped with forward slashes (`"/"`), e.g. `"/2\d\d/"` to match 2*XX* status codes.

Route response selection expressions

A route response is used for modeling a response from the backend to the client. For WebSocket APIs, a route response is optional. When defined, it signals to API Gateway that it should return a response to a client upon receiving a WebSocket message.

Evaluation of the *route response selection expression* produces a route response key. Eventually, this key will be used to choose from one of the `RouteResponses` associated with the API. However, currently only the `$default` key is supported.

API key selection expressions

This expression is evaluated when the service determines the given request should proceed only if the client provides a valid API key.

Currently the only two supported values are `$request.header.x-api-key` and `$context.authorizer.usageIdentifierKey`.

API mapping selection expressions

This expression is evaluated to determine which API stage is selected when a request is made using a custom domain.

Currently, the only supported value is `$request.basepath`.

WebSocket selection expression summary

The following table summarizes the use cases for selection expressions in WebSocket APIs:

Selection expression	Evaluates to key for	Notes	Example use case
`Api.RouteSelectionExpression`	`Route.RouteKey`	`$defaul` is supporte as a catch-all route.	Route WebSocket messages based on the context of a client request.
`Route.ModelSelectionExpression`	Key for `Route.RequestModels`	Optional If provided for non-proxy integration, model validation occurs. `$defaul` is supporte as a catch-all.	Perform request validation dynamically within the same route.

Selection expression	Evaluates to key for	Notes	Example use case
`Integration.TemplateSelectionExpression`	Key for `Integration.RequestTemplates`	Optional May be provided for non-proxy integration to manipul e incomin payloads `${reque .body.j nPath}` and static values are supporte . `$defau` is supporte as a catch-all.	Manipulate the caller's request based on dynamic properties of the request.

Selection expression	Evaluates to key for	Notes	Example use case
`Integration.IntegrationResponseSelectionExpression`	`IntegrationResponse.IntegrationResponseKey`	Optional May be provided for non-proxy integration. Acts as a pattern match for error messages (from Lambda) or status codes (from HTTP integrations). `$defaul` is required for non-proxy	Manipulate the response from the backend. Choose the action to occur based on the dynamic response of the backend (e.g., handling certain errors distinctly).

Selection expression	Evaluates to key for	Notes	Example use case
		integrations to act as the catch-all for successful response.	

Selection expression	Evaluates to key for	Notes	Example use case
`IntegrationResponse.TemplateSelectionExpression`	Key for `IntegrationResponse.ResponseTemplates`	Optional May be provided for non-proxy integration. $default is supporte.	In some cases, a dynamic property of the response may dictate different transformations within the same route and associated integration. `${request.body.jsonPath}` , `${integration.response.statuscode}` , `${integration.response.head`

Selection expression	Evaluates to key for	Notes	Example use case
			er.headerName} , ${integration.response.multivalueheader.headerName} , and static values are supported. $default is supported as a catch-all.

Selection expression	Evaluates to key for	Notes	Example use case
`Route.RouteResponseSelectionExpression`	`RouteResponse.RouteResponseKey`	Should be provided to initiate two-way commun tion for a WebSock route. Currentl , this value is restricted to `$defaul` only.	
`RouteResponse.ModelSelectionExpression`	Key for `RouteResponse.RequestModels`	Currentl unsuppo ed.	

Developing a WebSocket API in API Gateway

This section provides details about API Gateway capabilities that you need while you're developing your API Gateway APIs.

As you're developing your API Gateway API, you decide on a number of characteristics of your API. These characteristics depend on the use case of your API. For example, you might want to only allow certain clients to call your API, or you might want it to be available to everyone. You might want an API call to execute a Lambda function, make a database query, or call an application.

Topics

Create a WebSocket API in API Gateway

You can create a WebSocket API in the API Gateway console, by using the AWS CLI `create-api` command, or by using the `CreateApi` command in an AWS SDK. The following procedures show how to create a new WebSocket API.

> **Note**
>
> WebSocket APIs only support TLS 1.2. Earlier TLS versions are not supported.

Create a WebSocket API using AWS CLI commands

Creating a WebSocket API using the AWS CLI requires calling the `create-api` command as shown in the following example, which creates an API with the `$request.body.action` route selection expression:

```
aws apigatewayv2 --region us-east-1 create-api --name "myWebSocketApi3" --protocol-type
 WEBSOCKET --route-selection-expression '$request.body.action'
```

Example output:

```
{
    "ApiKeySelectionExpression": "$request.header.x-api-key",
    "Name": "myWebSocketApi3",
    "CreatedDate": "2018-11-15T06:23:51Z",
    "ProtocolType": "WEBSOCKET",
    "RouteSelectionExpression": "'$request.body.action'",
    "ApiId": "aabbccddee"
}
```

Create a WebSocket API using the API Gateway console

You can create a WebSocket API in the console by choosing the WebSocket protocol and giving the API a name.

> ⚠ **Important**
>
> Once you have created the API, you cannot change the protocol you have chosen for it. There is no way to convert a WebSocket API into a REST API or vice versa.

To create a WebSocket API using the API Gateway console

1. Sign in to the API Gateway console and choose **Create API**.
2. Under **WebSocket API**, choose **Build**. Only Regional endpoints are supported.
3. For **API name**, enter the name of your API.
4. For **Route selection expression**, enter a value. For example, `$request.body.action`.

   For more information about route selection expressions, see the section called “”.
5. Do one of the following:
   - Choose **Create blank API** to create an API with no routes.
   - Choose **Next** to attach routes to your API.

   You can attach routes after you create your API.

Working with routes for WebSocket APIs

In your WebSocket API, incoming JSON messages are directed to backend integrations based on routes that you configure. (Non-JSON messages are directed to a `$default` route that you configure.)

A *route* includes a *route key*, which is the value that is expected once a *route selection expression* is evaluated. The `routeSelectionExpression` is an attribute defined at the API level. It specifies a JSON property that is expected to be present in the message payload. For more information about route selection expressions, see the section called “”.

For example, if your JSON messages contain an `action` property and you want to perform different actions based on this property, your route selection expression might be `${request.body.action}`. Your routing table would specify which action to perform by matching the value of the `action` property against the custom route key values that you have defined in the table.

There are three predefined routes that can be used: `$connect`, `$disconnect`, and `$default`. In addition, you can create custom routes.

- API Gateway calls the `$connect` route when a persistent connection between the client and a WebSocket API is being initiated.
- API Gateway calls the `$disconnect` route when the client or the server disconnects from the API.
- API Gateway calls a custom route after the route selection expression is evaluated against the message if a matching route is found; the match determines which integration is invoked.
- API Gateway calls the `$default` route if the route selection expression cannot be evaluated against the message or if no matching route is found.

Route selection expressions

A *route selection expression* is evaluated when the service is selecting the route to follow for an incoming message. The service uses the route whose `routeKey` exactly matches the evaluated value. If none match and a route with the `$default` route key exists, that route is selected. If no routes match the evaluated value and there is no `$default` route, the service returns an error. For WebSocket-based APIs, the expression should be of the form `$request.body.`*`{path_to_body_element}`*.

For example, suppose you are sending the following JSON message:

```
{
    "service" : "chat",
    "action" : "join",
    "data" : {
        "room" : "room1234"
   }
}
```

You might want to select your API's behavior based on the `action` property. In that case, you might define the following route selection expression:

```
$request.body.action
```

In this example, `request.body` refers to your message's JSON payload, and `.action` is a [JSONPath](https://goessner.net/articles/JsonPath/) expression. You can use any JSON path expression after `request.body`, but keep in mind that the result will be stringified. For example, if your JSONPath expression returns an array of two elements, that will be presented as the string `"[item1, item2]"`. For this reason, it's a good practice to have your expression evaluate to a value and not an array or an object.

You can simply use a static value, or you can use multiple variables. The following table shows examples and their evaluated results against the preceding payload.

Expression	Evaluated result	Description
`$request.body.action`	`join`	An unwrapped variable
`${request.body.action}`	`join`	A wrapped variable
`${request.body.service}/${r`	`chat/join`	Multiple variables with

Expression	Evaluated result	Description
`equest.body.action}`		static values
`${request.body.action}-${request.body.invalidPath}`	`join-`	If the JSONPath is not found, the variable is resolved as "".
`action`	`action`	Static value
`\$default`	`$default`	Static value

The evaluated result is used to find a route. If there is a route with a matching route key, the route is selected to process the message. If no matching route is found, then API Gateway tries to find the `$default` route if available. If the `$default` route is not defined, then API Gateway returns an error.

Set up routes for a WebSocket API in API Gateway

When you first create a new WebSocket API, there are three predefined routes: `$connect`, `$disconnect`, and `$default`. You can create them by using the console, API, or AWS CLI. If desired, you can create custom routes. For more information, see the section called “About WebSocket APIs”.

> **Note**
>
> In the CLI, you can create routes before or after you create integrations, and you can reuse the same integration for multiple routes.

Create a route using the API Gateway console

To create a route using the API Gateway console

1. Sign in to the API Gateway console, choose the API, and choose **Routes**.
2. Choose **Create route**
3. For **Route key**, enter the route key name. You can create the predefined routes (`$connect`, `$disconnect`, and `$default`), or a custom route.

   > **Note**
   >
   > When you create a custom route, do not use the `$` prefix in the route key name. This prefix is reserved for predefined routes.

4. Select and configure the integration type for the route. For more information, see the section called "Set up a WebSocket API integration request using the API Gateway console".

Create a route using the AWS CLI

To create a route using the AWS CLI, call `create-route` as shown in the following example:

```
aws apigatewayv2 --region us-east-1 create-route --api-id aabbccddee --route-key
 $default
```

Example output:

```
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteKey": "$default",
    "RouteId": "1122334"
}
```

Specify route request settings for `$connect`

When you set up the `$connect` route for your API, the following optional settings are available to enable authorization for your API. For more information, see the section called "The `$connect` route".

- **Authorization**: If no authorization is needed, you can specify `NONE`. Otherwise, you can specify:
  - `AWS_IAM` to use standard AWS IAM policies to control access to your API.
  - `CUSTOM` to implement authorization for an API by specifying a Lambda authorizer function that you have previously created. The authorizer can reside in your own AWS account or a different AWS account. For more information about Lambda authorizers, see Use API Gateway Lambda authorizers.

    > **Note**
    >
    > In the API Gateway console, the `CUSTOM` setting is visible only after you have set up an authorizer function as described in the section called “Configure a Lambda authorizer using the console”.

  > **Important**
  >
  > The **Authorization** setting is applied to the entire API, not just the `$connect` route. The `$connect` route protects the other routes, because it is called on every connection.

- **API Key Required**: You can optionally require an API key for an API's `$connect` route. You can use API keys together with usage plans to control and track access to your APIs. For more information, see the section called “Usage plans”.

Set up the `$connect` route request using the API Gateway console

To set up the `$connect` route request for a WebSocket API using the API Gateway console:

1. Sign in to the API Gateway console, choose the API, and choose **Routes**.
2. Under **Routes**, choose `$connect`, or create a `$connect` route by following the section called “Create a route using the API Gateway console”.
3. In the **Route request settings** section, choose **Edit**.
4. For **Authorization**, select an authorization type.
5. To require an API for the `$connect` route, select **Require API key**.
6. Choose **Save changes**.

Set up route responses for a WebSocket API in API Gateway

WebSocket routes can be configured for two-way or one-way communication. API Gateway will not pass the backend response through to the route response, unless you set up a route response.

> **Note**
>
> You can only define the `$default` route response for WebSocket APIs. You can use an integration response to manipulate the response from a backend service. For more information, see the section called "Overview of integration responses".

You can configure route responses and response selection expressions by using the API Gateway console or the AWS CLI or an AWS SDK.

For more information about route response selection expressions, see the section called "".

Topics

- Set up a route response using the API Gateway console
- Set up a route response using the AWS CLI

Set up a route response using the API Gateway console

After you have created a WebSocket API and attached a proxy Lambda function to the default route, you can set up route response using the API Gateway console:

1. Sign in to the API Gateway console, choose a WebSocket API with a proxy Lambda function integration on the `$default` route.
2. Under **Routes**, choose the `$default` route.
3. Choose **Enable two-way communication**.
4. Choose **Deploy API**.
5. Deploy your API to a stage.

Use the following wscat command to connect to your API. For more information about `wscat`, see the section called "Use `wscat` to connect to a WebSocket API and send messages to it".

```
wscat -c wss://api-id.execute-api.us-east-2.amazonaws.com/test
```

Press the enter button to call the default route. The body of your Lambda function should return.

Set up a route response using the AWS CLI

To set up a route response for a WebSocket API using the AWS CLI, call the `create-route-response` command as shown in the following example. You can identify the API ID and route ID by calling `get-apis` and `get-routes`.

```
aws apigatewayv2 create-route-response \
    --api-id aabbccddee \
    --route-id 1122334  \
    --route-response-key '$default'
```

Example output:

```
{
    "RouteResponseId": "abcdef",
    "RouteResponseKey": "$default"
}
```

Setting up a `$connect` route that requires a WebSocket subprotocol

Clients can use the `Sec-WebSocket-Protocol` field to request a WebSocket subprotocol during the connection to your WebSocket API. You can set up an integration for the `$connect` route to allow connections only if a client requests a subprotocol that your API supports.

The following example Lambda function returns the `Sec-WebSocket-Protocol` header to clients. The function establishes a connection to your API only if the client specifies the `myprotocol` subprotocol.

For an AWS CloudFormation template that creates this example API and Lambda proxy integration, see `ws-subprotocol.yaml`.

```
export const handler = async (event) => {
    if (event.headers != undefined) {
        const headers = toLowerCaseProperties(event.headers);

        if (headers['sec-websocket-protocol'] != undefined) {
            const subprotocolHeader = headers['sec-websocket-protocol'];
            const subprotocols = subprotocolHeader.split(',');
            if (subprotocols.indexOf('myprotocol') >= 0) {
                const response = {
                    statusCode: 200,
                    headers: {
                        "Sec-WebSocket-Protocol" : "myprotocol"
                    }
                };
                return response;
            }
        }
    }

    const response = {
        statusCode: 400
    };

    return response;
};

function toLowerCaseProperties(obj) {
    var wrapper = {};
    for (var key in obj) {
        wrapper[key.toLowerCase()] = obj[key];
    }
    return wrapper;
}
```

You can use wscat to test that your API allows connections only if a client requests a subprotocol that your API supports. The following commands use the `-s` flag to specify subprotocols during the connection.

The following command attempts a connection with an unsupported subprotocol. Because the client specified the `chat1` subprotocol, the Lambda integration returns a 400 error, and the connection is unsuccessful.

```
wscat -c wss://api-id.execute-api.region.amazonaws.com/beta -s chat1
error: Unexpected server response: 400
```

The following command includes a supported subprotocol in the connection request. The Lambda integration allows the connection.

```
wscat -c wss://api-id.execute-api.region.amazonaws.com/beta -s chat1,myprotocol
connected (press CTRL+C to quit)
```

To learn more about invoking WebSocket APIs, see Invoking a WebSocket API.

Controlling and managing access to a WebSocket API in API Gateway

API Gateway supports multiple mechanisms for controlling and managing access to your WebSocket API.

You can use the following mechanisms for authentication and authorization:

- **Standard AWS IAM roles and policies** offer flexible and robust access controls. You can use IAM roles and policies for controlling who can create and manage your APIs, as well as who can invoke them. For more information, see Using IAM authorization.
- **IAM tags** can be used together with IAM policies to control access. For more information, see Using tags to control access to API Gateway REST API resources.
- **Lambda authorizers** are Lambda functions that control access to APIs. For more information, see Creating a Lambda REQUEST authorizer function.

Topics

Using IAM authorization

IAM authorization in WebSocket APIs is similar to that for REST APIs, with the following exceptions:

- The `execute-api` action supports `ManageConnections` in addition to existing actions (`Invoke`, `InvalidateCache`). `ManageConnections` controls access to the @connections API.
- WebSocket routes use a different ARN format:

  ```
  arn:aws:execute-api:region:account-id:api-id/stage-name/route-key
  ```

- The `@connections` API uses the same ARN format as REST APIs:

  ```
  arn:aws:execute-api:region:account-id:api-id/stage-name/POST/@connections
  ```

Important

When you use IAM authorization, you must sign requests with Signature Version 4 (SigV4).

For example, you could set up the following policy to the client. This example allows everyone to send a message (`Invoke`) for all routes except for a secret route in the `prod` stage and prevents everyone from sending a message back to connected clients (`ManageConnections`) for all stages.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/prod/*"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/prod/secret"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:ManageConnections"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*"
      ]
    }
  ]
}
```

Creating a Lambda REQUEST authorizer function

A Lambda authorizer function in WebSocket APIs is similar to that for REST APIs, with the following exceptions:

- You can only use a Lambda authorizer function for the `$connect` route.
- You cannot use path variables (`event.pathParameters`), because the path is fixed.
- `event.methodArn` is different from its REST API equivalent, because it has no HTTP method. In the case of `$connect`, `methodArn` ends with `"$connect"`:

  ```
  arn:aws:execute-api:region:account-id:api-id/stage-name/$connect
  ```

- The context variables in `event.requestContext` are different from those for REST APIs.

The following example shows an input to a `REQUEST` authorizer for a WebSocket API:

```
{
    "type": "REQUEST",
    "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/default/
$connect",
    "headers": {
        "Connection": "upgrade",
        "content-length": "0",
        "HeaderAuth1": "headerValue1",
        "Host": "abcdef123.execute-api.us-east-1.amazonaws.com",
        "Sec-WebSocket-Extensions": "permessage-deflate; client_max_window_bits",
        "Sec-WebSocket-Key": "...",
        "Sec-WebSocket-Version": "13",
        "Upgrade": "websocket",
        "X-Amzn-Trace-Id": "...",
        "X-Forwarded-For": "...",
        "X-Forwarded-Port": "443",
        "X-Forwarded-Proto": "https"
    },
    "multiValueHeaders": {
        "Connection": [
            "upgrade"
        ],
        "content-length": [
            "0"
        ],
```

```
        "HeaderAuth1": [
            "headerValue1"
        ],
        "Host": [
            "abcdef123.execute-api.us-east-1.amazonaws.com"
        ],
        "Sec-WebSocket-Extensions": [
            "permessage-deflate; client_max_window_bits"
        ],
        "Sec-WebSocket-Key": [
            "..."
        ],
        "Sec-WebSocket-Version": [
            "13"
        ],
        "Upgrade": [
            "websocket"
        ],
        "X-Amzn-Trace-Id": [
            "..."
        ],
        "X-Forwarded-For": [
            "..."
        ],
        "X-Forwarded-Port": [
            "443"
        ],
        "X-Forwarded-Proto": [
            "https"
        ]
    },
    "queryStringParameters": {
        "QueryString1": "queryValue1"
    },
    "multiValueQueryStringParameters": {
        "QueryString1": [
            "queryValue1"
        ]
    },
    "stageVariables": {},
    "requestContext": {
        "routeKey": "$connect",
        "eventType": "CONNECT",
        "extendedRequestId": "...",
        "requestTime": "19/Jan/2023:21:13:26 +0000",
        "messageDirection": "IN",
        "stage": "default",
        "connectedAt": 1674162806344,
        "requestTimeEpoch": 1674162806345,
        "identity": {
            "sourceIp": "..."
        },
        "requestId": "...",
        "domainName": "abcdef123.execute-api.us-east-1.amazonaws.com",
        "connectionId": "...",
        "apiId": "abcdef123"
    }
}
```

The following example Lambda authorizer function is a WebSocket version of the Lambda authorizer function for REST APIs in the section called “Create a Lambda authorizer function in the Lambda console”:

Node.js

```
  // A simple REQUEST authorizer example to demonstrate how to use request
  // parameters to allow or deny a request. In this example, a request is
  // authorized if the client-supplied HeaderAuth1 header and QueryString1 query
parameter
  // in the request context match the specified values of
  // of 'headerValue1' and 'queryValue1' respectively.
           export const handler = function(event, context, callback) {
   console.log('Received event:', JSON.stringify(event, null, 2));

  // Retrieve request parameters from the Lambda function input:
  var headers = event.headers;
  var queryStringParameters = event.queryStringParameters;
  var stageVariables = event.stageVariables;
  var requestContext = event.requestContext;

  // Parse the input for the parameter values
  var tmp = event.methodArn.split(':');
  var apiGatewayArnTmp = tmp[5].split('/');
  var awsAccountId = tmp[4];
  var region = tmp[3];
  var ApiId = apiGatewayArnTmp[0];
  var stage = apiGatewayArnTmp[1];
```

```
   var route = apiGatewayArnTmp[2];

   // Perform authorization to return the Allow policy for correct parameters and
   // the 'Unauthorized' error, otherwise.
   var authResponse = {};
   var condition = {};
    condition.IpAddress = {};

   if (headers.HeaderAuth1 === "headerValue1"
       && queryStringParameters.QueryString1 === "queryValue1") {
        callback(null, generateAllow('me', event.methodArn));
    }  else {
        callback("Unauthorized");
    }
}

// Helper function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
   // Required output:
   var authResponse = {};
    authResponse.principalId = principalId;
   if (effect && resource) {
       var policyDocument = {};
        policyDocument.Version = '2012-10-17'; // default version
       policyDocument.Statement = [];
       var statementOne = {};
        statementOne.Action = 'execute-api:Invoke'; // default action
       statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }
   // Optional output with custom properties of the String, Number or Boolean type.
   authResponse.context = {
       "stringKey": "stringval",
       "numberKey": 123,
       "booleanKey": true
    };
   return authResponse;
}

var generateAllow = function(principalId, resource) {
   return generatePolicy(principalId, 'Allow', resource);
}
```

```
var generateDeny = function(principalId, resource) {
   return generatePolicy(principalId, 'Deny', resource);
}
```

Python

```
# A simple REQUEST authorizer example to demonstrate how to use request
# parameters to allow or deny a request. In this example, a request is
# authorized if the client-supplied HeaderAuth1 header and QueryString1 query
 parameter
# in the request context match the specified values of
# of 'headerValue1' and 'queryValue1' respectively.

import json


def lambda_handler(event, context):
    print(event)

    # Retrieve request parameters from the Lambda function input:
    headers = event['headers']
    queryStringParameters = event['queryStringParameters']
    stageVariables = event['stageVariables']
    requestContext = event['requestContext']

    # Parse the input for the parameter values
    tmp = event['methodArn'].split(':')
    apiGatewayArnTmp = tmp[5].split('/')
    awsAccountId = tmp[4]
    region = tmp[3]
    ApiId = apiGatewayArnTmp[0]
    stage = apiGatewayArnTmp[1]
    route = apiGatewayArnTmp[2]

    # Perform authorization to return the Allow policy for correct parameters
    # and the 'Unauthorized' error, otherwise.

    authResponse = {}
    condition = {}
    condition['IpAddress'] = {}

    if (headers['HeaderAuth1'] ==
            "headerValue1" and queryStringParameters["QueryString1"] ==
 "queryValue1"):
        response = generateAllow('me', event['methodArn'])
        print('authorized')
        return json.loads(response)
    else:
        print('unauthorized')
        return 'unauthorized'

    # Help function to generate IAM policy


def generatePolicy(principalId, effect, resource):
    authResponse = {}
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument

    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }

    authResponse_JSON = json.dumps(authResponse)

    return authResponse_JSON


def generateAllow(principalId, resource):
    return generatePolicy(principalId, 'Allow', resource)


def generateDeny(principalId, resource):
    return generatePolicy(principalId, 'Deny', resource)
```

To configure the preceding Lambda function as a REQUEST authorizer function for a WebSocket API, follow the same procedure as for REST APIs.

To configure the $connect route to use this Lambda authorizer in the console, select or create the $connect route. In the **Route request settings** section, choose **Edit**. Select your authorizer in the **Authorization** dropdown menu, and then choose **Save changes**.

To test the authorizer, you need to create a new connection. Changing authorizer in $connect doesn't affect the already connected client. When you connect to your WebSocket API, you need to provide values for any configured identity sources. For example, you can connect by sending a valid query string and header using wscat as in the following example:

```
wscat -c 'wss://myapi.execute-api.us-east-1.amazonaws.com/beta?
QueryString1=queryValue1' -H HeaderAuth1:headerValue1
```

If you attempt to connect without a valid identity value, you'll receive a 401 response:

```
wscat -c wss://myapi.execute-api.us-east-1.amazonaws.com/beta
error: Unexpected server response: 401
```

Setting up WebSocket API integrations

After setting up an API route, you must integrate it with an endpoint in the backend. A backend endpoint is also referred to as an integration endpoint and can be a Lambda function, an HTTP endpoint, or an AWS service action. The API integration has an integration request and an integration response.

In this section, you can learn how to set up integration requests and integration responses for your WebSocket API.

Topics

Setting up a WebSocket API integration request in API Gateway

Setting up an integration request involves the following:

- Choosing a route key to integrate to the backend.
- Specifying the backend endpoint to invoke. WebSocket APIs support the following integration types:
  - `AWS_PROXY`
  - `AWS`
  - `HTTP_PROXY`
  - `HTTP`
  - `MOCK`

  For more information about integration types, see IntegrationType in the API Gateway V2 REST API.
- Configuring how to transform the route request data, if necessary, into integration request data by specifying one or more request templates.

Set up a WebSocket API integration request using the API Gateway console

To add an integration request to a route in a WebSocket API using the API Gateway console

1. Sign in to the API Gateway console, choose the API, and choose **Routes**.
2. Under **Routes**, choose the route.
3. Choose the **Integration request** tab, and then in the **Integration request settings** section, choose **Edit**.
4. For **Integration type**, select one of the following:
   - Choose **Lambda function** only if your API will be integrated with an AWS Lambda function that you have already created in this account or in another account.

     To create a new Lambda function in AWS Lambda, to set a resource permission on the Lambda function, or to perform any other Lambda service actions, choose **AWS Service** instead.
   - Choose **HTTP** if your API will be integrated with an existing HTTP endpoint. For more information, see Set up HTTP integrations in API Gateway.
   - Choose **Mock** if you want to generate API responses from API Gateway directly, without the need for an integration backend. For more information, see Set up mock integrations in API Gateway.

- Choose **AWS service** if your API will be integrated with an AWS service.
- Choose **VPC link** if your API will use a VpcLink as a private integration endpoint. For more information, see Set up API Gateway private integrations.

5. If you chose **Lambda function**, do the following:

   a. For **Use Lambda proxy integration**, choose the check box if you intend to use Lambda proxy integration or cross-account Lambda proxy integration.

   b. For **Lambda function**, specify the function in one of the following ways:

      - If your Lambda function is in the same account, enter the function name and then select the function from the dropdown list.

        Note

        The function name can optionally include its alias or version specification, as in `HelloWorld`, `HelloWorld:1`, or `HelloWorld:alpha`.

      - If the function is in a different account, enter the ARN for the function.

   c. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.

6. If you chose **HTTP**, follow the instructions in step 4 of the section called “ Set up integration request using the console”.
7. If you chose **Mock**, proceed to the **Request Templates** step.
8. If you chose **AWS service**, follow the instructions in step 6 of the section called “ Set up integration request using the console”.
9. If you chose **VPC link**, do the following:

   a. For **VPC proxy integration**, choose the check box if you want your requests to be proxied to your `VPCLink`'s endpoint.

   b. For **HTTP method**, choose the HTTP method type that most closely matches the method in the HTTP backend.

   c. From the **VPC link** dropdown list, select a VPC link. You can select `[Use Stage Variables]` and enter **`${stageVariables.vpcLinkId}`** in the text box below the list.

    You can define the `vpcLinkId` stage variable after deploying the API to a stage and set its value to the ID of the `VpcLink`.

    d. For **Endpoint URL**, enter the URL of the HTTP backend you want this integration to use.

    e. To use the default timeout value of 29 seconds, keep **Default timeout** turned on. To set a custom timeout, choose **Default timeout** and enter a timeout value between `50` and `29000` milliseconds.

10. Choose **Save changes**.

11. Under **Request templates**, do the following:

    a. To enter a **Template selection expression**, under **Request templates**, choose **Edit**.

    b. Enter a **Template selection expression**. Use an expression that API Gateway looks for in the message payload. If it is found, it is evaluated, and the result is a template key value that is used to select the data mapping template to be applied to the data in the message payload. You create the data mapping template in the next step. Choose **Edit** to save your changes.

    c. Choose **Create template** to create the data mapping template. For **Template key**, enter a template key value that is used to select the data mapping template to be applied to the data in the message payload. Then, enter a mapping template. Choose **Create template**.

    For information about template selection expressions, see [the section called “Template selection expressions”](#).

Set up an integration request using the AWS CLI

You can set up an integration request for a route in a WebSocket API by using the AWS CLI as in the following example, which creates a mock integration:

1. Create a file named `integration-params.json`, with the following contents:

```
{"PassthroughBehavior": "WHEN_NO_MATCH", "TimeoutInMillis": 29000,
 "ConnectionType": "INTERNET", "RequestTemplates": {"application/json":
 "{\"statusCode\":200}"}, "IntegrationType": "MOCK"}
```

2. Run the [`create-integration`](#) command as shown in the following example:

```
aws apigatewayv2 --region us-east-1 create-integration --api-id aabbccddee --cli-input-json file://integration-params.json
```

Following is sample output for this example:

```
{
    "PassthroughBehavior": "WHEN_NO_MATCH",
    "TimeoutInMillis": 29000,
    "ConnectionType": "INTERNET",
    "IntegrationResponseSelectionExpression": "${response.statuscode}",
    "RequestTemplates": {
        "application/json": "{\"statusCode\":200}"
    },
    "IntegrationId": "0abcdef",
    "IntegrationType": "MOCK"
}
```

Alternatively, you can set up an integration request for a proxy integration by using the AWS CLI as in the following example:

1. Create a Lambda function in the Lambda console and give it a basic Lambda execution role.
2. Execute the `create-integration` command as in the following example:

```
aws apigatewayv2 create-integration --api-id aabbccddee --integration-type
 AWS_PROXY --integration-method POST --integration-uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:123412341234:function:simpleproxy-echo-e2e/invocations
```

Following is sample output for this example:

```
{
    "PassthroughBehavior": "WHEN_NO_MATCH",
    "IntegrationMethod": "POST",
    "TimeoutInMillis": 29000,
    "ConnectionType": "INTERNET",
    "IntegrationUri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123412341234:function:simpleproxy-echo-e2e/invocations",
    "IntegrationId": "abcdefg",
    "IntegrationType": "AWS_PROXY"
}
```

Input format of a Lambda function for proxy integration for WebSocket APIs

In Lambda proxy integration, API Gateway maps the entire client request to the input `event` parameter of the backend Lambda function. The following example shows the structure of the input event from the `$connect` route and the input event from the `$disconnect` route that API Gateway sends to a Lambda proxy integration.

Input from the $connect route

```
{
    headers: {
      Host: 'abcd123.execute-api.us-east-1.amazonaws.com',
      'Sec-WebSocket-Extensions': 'permessage-deflate; client_max_window_bits',
      'Sec-WebSocket-Key': '...',
      'Sec-WebSocket-Version': '13',
      'X-Amzn-Trace-Id': '...',
      'X-Forwarded-For': '192.0.2.1',
      'X-Forwarded-Port': '443',
      'X-Forwarded-Proto': 'https'
    },
    multiValueHeaders: {
      Host: [ 'abcd123.execute-api.us-east-1.amazonaws.com' ],
      'Sec-WebSocket-Extensions': [ 'permessage-deflate; client_max_window_bits' ],
      'Sec-WebSocket-Key': [ '...' ],
      'Sec-WebSocket-Version': [ '13' ],
      'X-Amzn-Trace-Id': [ '...' ],
      'X-Forwarded-For': [ '192.0.2.1' ],
      'X-Forwarded-Port': [ '443' ],
      'X-Forwarded-Proto': [ 'https' ]
    },
    requestContext: {
      routeKey: '$connect',
      eventType: 'CONNECT',
      extendedRequestId: 'ABCD1234=',
      requestTime: '09/Feb/2024:18:11:43 +0000',
      messageDirection: 'IN',
      stage: 'prod',
      connectedAt: 1707502303419,
      requestTimeEpoch: 1707502303420,
      identity: { sourceIp: '192.0.2.1' },
      requestId: 'ABCD1234=',
   domainName: 'abcd1234.execute-api.us-east-1.amazonaws.com',
   connectionId: 'AAAA1234=',
   apiId: 'abcd1234'
  },
  isBase64Encoded: false
 }
```

Input from the $disconnect route

```
{
  headers: {
   Host: 'abcd1234.execute-api.us-east-1.amazonaws.com',
   'x-api-key': '',
   'X-Forwarded-For': '',
   'x-restapi': ''
  },
  multiValueHeaders: {
   Host: [ 'abcd1234.execute-api.us-east-1.amazonaws.com' ],
   'x-api-key': [ '' ],
   'X-Forwarded-For': [ '' ],
   'x-restapi': [ '' ]
  },
  requestContext: {
   routeKey: '$disconnect',
   disconnectStatusCode: 1005,
   eventType: 'DISCONNECT',
   extendedRequestId: 'ABCD1234=',
   requestTime: '09/Feb/2024:18:23:28 +0000',
   messageDirection: 'IN',
   disconnectReason: 'Client-side close frame status not set',
   stage: 'prod',
   connectedAt: 1707503007396,
   requestTimeEpoch: 1707503008941,
   identity: { sourceIp: '192.0.2.1' },
   requestId: 'ABCD1234=',
   domainName: 'abcd1234.execute-api.us-east-1.amazonaws.com',
   connectionId: 'AAAA1234=',
   apiId: 'abcd1234'
  },
  isBase64Encoded: false
 }
```

Setting up a WebSocket API integration responses in API Gateway

Topics

- Overview of integration responses
- Integration responses for two-way communication
- Set up an integration response using the API Gateway console
- Set up an integration response using the AWS CLI

Overview of integration responses

API Gateway's integration response is a way of modeling and manipulating the response from a backend service. There are some differences in setup of a REST API versus a WebSocket API integration response, but conceptually the behavior is the same.

WebSocket routes can be configured for two-way or one-way communication.

- When a route is configured for two-way communication, an integration response allows you to configure transformations on the returned message payload, similar to integration responses for REST APIs.
- If a route is configured for one-way communication, then regardless of any integration response configuration, no response will be returned over the WebSocket channel after the message is processed.

API Gateway will not pass the backend response through to the route response, unless you set up a route response. To learn about setting up a route response, see the section called “Set up WebSocket API route responses”.

Integration responses for two-way communication

Integrations can be divided into *proxy* integrations and *non-proxy* integrations.

> **Important**
>
> For *proxy integrations*, API Gateway automatically passes the backend output to the caller as the complete payload. There is no integration response.

For *non-proxy integrations*, you must set up at least one integration response:

- Ideally, one of your integration responses should act as a catch-all when no explicit choice can be made. This default case is represented by setting an integration response key of `$default`.
- In all other cases, the integration response key functions as a regular expression. It should follow a format of `"/expression/"`.

For non-proxy HTTP integrations:

- API Gateway will attempt to match the HTTP status code of the backend response. The integration response key will function as a regular expression in this case. If a match cannot be found, then `$default` is chosen as the integration response.
- The template selection expression, as described above, functions identically. For example:
  - `/2\d\d/`: Receive and transform successful responses
  - `/4\d\d/`: Receive and transform bad request errors
  - `$default`: Receive and transform all unexpected responses

For more information about template selection expressions, see the section called “Template selection expressions”.

Set up an integration response using the API Gateway console

To set up a route integration response for a WebSocket API using the API Gateway console:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your WebSocket API and choose your route.
3. Choose the **Integration request** tab, and then in the **Integration response settings** section, choose **Create integration response**.
4. For **Response key**, enter a value that will be found in the response key in the outgoing message after evaluating the response selection expression. For instance, you can enter **`/4\d\d/`** to receive and transform bad request errors or enter **`$default`** to receive and transform all responses that match the template selection expression.
5. For **Template selection expression**, enter a selection expression to evaluate the outgoing message.
6. Choose **Create response**.
7. You can also define a mapping template to configure transformations of your returned message payload. Choose **Create template**.

8. Enter a key name. If you are choosing the default template selection expression, enter **\$default**.
9. For **Response template**, enter your mapping template in the code editor.
10. Choose **Create template**.
11. Choose **Deploy API** to deploy your API.

Use the following wscat command to connect to your API. For more information about `wscat`, see the section called "Use `wscat` to connect to a WebSocket API and send messages to it".

```
wscat -c wss://api-id.execute-api.us-east-2.amazonaws.com/test
```

When you call your route, the returned message payload should return.

Set up an integration response using the AWS CLI

To set up an integration response for a WebSocket API using the AWS CLI call the `create-integration-response` command. The following CLI command shows an example of creating a `$default` integration response:

```
aws apigatewayv2 create-integration-response \
    --api-id vaz7da96z6 \
    --integration-id a1b2c3 \
    --integration-response-key '$default'
```

Request validation

You can configure API Gateway to perform validation on a route request before proceeding with the integration request. If the validation fails, API Gateway fails the request without calling your backend, sends a "Bad request body" gateway response to the client, and publishes the validation results in CloudWatch Logs. Using validation this way reduces unnecessary calls to your API backend.

Model selection expressions

You can use a model selection expression to dynamically validate requests within the same route. Model validation occurs if you provide a model selection expression for either proxy or non-proxy integrations. You might need to define the `$default` model as a fallback when no matching model is found. If there is no matching model and `$default` isn't defined, the validation fails. The selection expression looks like `Route.ModelSelectionExpression` and evaluates to the key for `Route.RequestModels`.

When you define a route for a WebSocket API, you can optionally specify a *model selection expression*. This expression is evaluated to select the model to be used for body validation when a request is received. The expression evaluates to one of the entries in a route's `requestmodels`.

A model is expressed as a JSON schema and describes the data structure of the request body. The nature of this selection expression enables you to dynamically choose the model to validate against at runtime for a particular route. For information about how to create a model, see the section called "Understanding data models".

Set up request validation using the API Gateway console

The following example shows you how to set up request validation on a route.

First, you create a model, and then you create a route. Next, you configure request validation on the route you just created. Lastly, you deploy and test your API. To complete this tutorial, you need a WebSocket API with `$request.body.action` as the route selection expression and an integration endpoint for your new route.

You also need `wscat` to connect to your API. For more information, see the section called "Use `wscat` to connect to a WebSocket API and send messages to it".

To create a model

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose a WebSocket API.
3. In the main navigation pane, choose **Models**.
4. Choose **Create model**.
5. For **Name**, enter **`emailModel`**.
6. For **Content type**, enter **`application/json`**.
7. For **Model schema**, enter the following model:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type" : "object",
    "required" : [ "address"],
    "properties" : {
        "address": {
            "type": "string"
        }
    }
}
```

This model requires that the request contains an email address.

8. Choose **Save**.

In this step, you create a route for your WebSocket API.

To create a route

1. In the main navigation pane, choose **Routes**.
2. Choose **Create route**.
3. For **Route key**, enter **`sendMessage`**.
4. Choose an integration type and specify an integration endpoint. For more information see [the section called "Integrations"](#).
5. Choose **Create route**.

In this step, you set up request validation for the `sendMessage` route.

To set up request validation

1. On the **Route request** tab, under **Route request settings**, choose **Edit**.
2. For **Model selection expression**, enter **`${request.body.messageType}`**.

   API Gateway uses the `messageType` property to validate the incoming request.
3. Choose **Add request model**.
4. For **Model key**, enter **`email`**.
5. For **Model**, choose **emailModel**.

   API Gateway validates incoming messages with the `messageType` property set to `email` against this model.

> **Note**
>
> If API Gateway can't match the model selection expression to a model key, then it selects the `$default` model. If there is no `$default` model, then the validation fails. For production APIs, we recommend that you create a `$default` model.

6. Choose **Save changes**.

In this step, you deploy and test your API.

To deploy and test your API

1. Choose **Deploy API**.
2. Choose the desired stage from the dropdown list or enter the name of a new stage.
3. Choose **Deploy**.
4. In the main navigation pane, choose **Stages**.
5. Copy your API's WebSocket URL. The URL should look like `wss://`*`abcdef123`*`.execute-api.`*`us-east-2`*`.amazonaws.com/production`.
6. Open a new terminal and run the **wscat** command with the following parameters.

```
wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
```

```
Connected (press CTRL+C to quit)
```

7. Use the following command to test your API.

```
{"action": "sendMessage", "messageType": "email"}
```

```
{"message": "Invalid request body", "connectionId":"ABCD1=234",
 "requestId":"EFGH="}
```

API Gateway will fail the request.

Use the next command to send a valid request to your API.

```
{"action": "sendMessage", "messageType": "email", "address":
 "mary_major@example.com"}
```

Setting up data transformations for WebSocket APIs

In API Gateway, a WebSocket API's method request can take a payload in a different format from the corresponding integration request payload, as required in the backend. Similarly, the backend may return an integration response payload different from the method response payload, as expected by the frontend.

API Gateway lets you use mapping templates to map the payload from a method request to the corresponding integration request and from an integration response to the corresponding method response. You specify a template selection expression to determine which template to use to perform the necessary data transformations.

You can use data mappings to map data from a route request to a backend integration. To learn more, see the section called “Data mapping”.

Mapping templates and models

A *mapping template* is a script expressed in Velocity Template Language (VTL) and applied to the payload using JSONPath expressions. For more information about API Gateway mapping templates, see Understanding mapping templates.

The payload can have a *data model* according to the JSON schema draft 4. You do not have to define a model to create a mapping template. However, a model can help you create a template because API Gateway generates a template blueprint based on a provided model. For more information about API Gateway models, see Understanding data models.

Template selection expressions

To transform a payload with a mapping template, you specify a WebSocket API template selection expression in an integration request or integration response. This expression is evaluated to determine the input or output template (if any) to use to transform either the request body into the integration request body (via an input template) or the integration response body to the route response body (via an output template).

`Integration.TemplateSelectionExpression` supports `${request.body.jsonPath}` and static values.

IntegrationResponse.TemplateSelectionExpression supports ${request.body.jsonPath}, ${integration.response.statuscode}, ${integration.response.header.headerName}, ${integration.response.multivalueheader.headerName}, and static values.

Integration response selection expressions

When you set up an integration response for a WebSocket API, you can optionally specify an integration response selection expression. This expression determines what IntegrationResponse should be selected when an integration returns. The value of this expression is currently restricted by API Gateway, as defined below. Realize that this expression is only relevant for *non-proxy integrations*; a proxy integration simply passes the response payload back to the caller without modeling or modification.

Unlike the other preceding selection expressions, this expression currently supports a *pattern-matching* format. The expression should be wrapped with forward slashes.

Currently the value is fixed depending on the integrationType:

- For Lambda-based integrations, it is $integration.response.body.errorMessage.
- For HTTP and MOCK integrations, it is $integration.response.statuscode.
- For HTTP_PROXY and AWS_PROXY, the expression isn't utilized because you're requesting that the payload pass through to the caller.

Setting up data mapping for WebSocket APIs

Data mapping enables you to map data from a route request to a backend integration.

> **Note**
>
> Data mapping for WebSocket APIs isn't supported in the AWS Management Console. You must use the AWS CLI, AWS CloudFormation, or an SDK to configure data mapping.

Topics

Map route request data to integration request parameters

Integration request parameters can be mapped from any defined route request parameters, the request body, `context` or `stage` variables, and static values.

In the following table, *PARAM_NAME* is the name of a route request parameter of the given parameter type. It must match the regular expression `'^[a-zA-Z0-9._$-]+$]'`. *JSONPath_EXPRESSION* is a JSONPath expression for a JSON field of the request body.

Integration request data mapping expressions

Mapped data source	Mapping expression
Request query string (supported only for the `$connect` route)	`route.request.querystring.` *PARAM_NAME*
Request header (supported only for the `$connect` route)	`route.request.header.` *PARAM_NAME*
Multi-value request query string (supported only for the `$connect` route)	`route.request.multivaluequerystring.` *PARAM_NAME*
Multi-value request header (supported only for the `$connect` route)	`route.request.multivalueheader.` *PARAM_NAME*
Request body	`route.request.body.` *JSONPath_EXPRESSION*
Stage variables	`stageVariables.` *VARIABLE_NAME*
Context variables	`context.`*VARIABLE_NAME* that must be one of the supported context variables.
Static value	*'STATIC_VALUE'* . The *STATIC_VALUE* is a string literal and must be enclosed in single quotes.

Examples

The following AWS CLI examples configure data mappings. For an example AWS CloudFormation template, see websocket-data-mapping.yaml.

Map a client's connectionId to a header in an integration request

The following example command maps a client's `connectionId` to a `connectionId` header in the request to a backend integration.

```
aws apigatewayv2 update-integration \
    --integration-id abc123 \
    --api-id a1b2c3d4 \
    --request-parameters
 'integration.request.header.connectionId'='context.connectionId'
```

Map a query string parameter to a header in an integration request

The following example commands map an `authToken` query string parameter to an `authToken` header in the integration request.

First, add the `authToken` query string parameter to the route's request parameters.

```
aws apigatewayv2 update-route --route-id 0abcdef \
    --api-id a1b2c3d4 \
    --request-parameters '{"route.request.querystring.authToken": {"Required": false}}'
```

Next, map the query string parameter to the `authToken` header in the request to the backend integration.

```
aws apigatewayv2 update-integration \
    --integration-id abc123 \
    --api-id a1b2c3d4 \
    --request-parameters
 'integration.request.header.authToken'='route.request.querystring.authToken'
```

If necessary, delete the `authToken` query string parameter from the route's request parameters.

```
aws apigatewayv2 delete-route-request-parameter \
    --route-id 0abcdef \
    --api-id a1b2c3d4 \
    --request-parameter-key 'route.request.querystring.authToken'
```

API Gateway WebSocket API mapping template reference

This section summarizes the set of variables that are currently supported for WebSocket APIs in API Gateway.

Parameter	Description
`$context.connectionId`	A unique ID for the connection that can be used to make a callback to the client.
`$context.connectedAt`	The Epoch-formatted connection time.
`$context.domainName`	A domain name for the WebSocket API. This can be used to make a callback to the client (instead of a hard-coded value).
`$context.eventType`	The event type: `CONNECT`, `MESSAGE`, or `DISCONNECT` .
`$context.messageId`	A unique server-side ID for a message. Available only when the `$context.eventType` is `MESSAGE`.
`$context.routeKey`	The selected route key.
`$context.requestId`	Same as `$context.extendedRequestId` .
`$context.extendedRequestId`	An automatically generated ID for the API call, which contains more useful information for debugging/troubleshooting.
`$context.apiId`	The identifier API Gateway assigns to your API.
`$context.authorizer.principalId`	The principal user identification associated with the token sent by the client and returned from an API Gateway Lambda authorizer

Parameter	Description
	(formerly known as a custom authorizer) Lambda function.

`$context.authorizer.` *`property`*	The stringified value of the specified key-value pair of the `context` map returned from an API Gateway Lambda authorizer function. For example, if the authorizer returns the following `context` map:
	`"context" : {`
	`  "key": "value",`
	`  "numKey": 1,`
	`  "boolKey": true`
	`}`
	calling `$context.authorizer.key` returns the `"value"` string, calling `$context.authorizer.numKey` returns the `"1"` string, and calling `$context.authorizer.boolKey` returns the `"true"` string.

`$context.error.messageString`	The quoted value of `$context.error.message` , namely `"$context.error.message"` .
`$context.error.validationErrorString`	A string containing a detailed validation error message.
`$context.identity.accountId`	The AWS account ID associated with the request.
`$context.identity.apiKey`	The API owner key associated with key-enabled API request.
`$context.identity.apiKeyId`	The API key ID associated with the key-enabled API request

Parameter	Description
`$context.identity.caller`	The principal identifier of the caller making the request.
`$context.identity.cognitoAuthenticationProvider`	A comma-separated list of the Amazon Cognito authentication providers used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For example, for an identity from an Amazon Cognito user pool, `cognito-idp.` *`region`*`.amazonaws.com/` *`user_pool_id`* `,cognito-idp.` *`region`*`.amazonaws.com/` *`user_pool_id`* `:CognitoSignIn:` *`token subject claim`* For information, see Using Federated Identities in the *Amazon Cognito Developer Guide*.
`$context.identity.cognitoAuthenticationType`	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials. Possible values include `authenticated` for authenticated identities and `unauthenticated` for unauthenticated identities.
`$context.identity.cognitoIdentityId`	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.cognitoIdentityPoolId`	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.

Parameter	Description
`$context.identity.sourceIp`	The source IP address of the immediate TCP connection making the request to API Gateway endpoint.
`$context.identity.user`	The principal identifier of the user making the request.
`$context.identity.userAgent`	The User Agent of the API caller.
`$context.identity.userArn`	The Amazon Resource Name (ARN) of the effective user identified after authentication.
`$context.requestTime`	The CLF-formatted request time (`dd/MMM/yyyy:HH:mm:ss +-hhmm`).
`$context.requestTimeEpoch`	The Epoch-formatted request time, in milliseconds.
`$context.stage`	The deployment stage of the API call (for example, Beta or Prod).
`$context.status`	The response status.
`$input.body`	Returns the raw payload as a string.
`$input.json(x)`	This function evaluates a JSONPath expression and returns the results as a JSON string. For example, `$input.json('$.pets')` will return a JSON string representing the pets structure. For more information about JSONPath, see JSONPath or JSONPath for Java.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td><code>$input.path(x)</code></td><td>Takes a JSONPath expression string (x) and returns a JSON object representation of the result. This allows you to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL).

For example, if the expression <code>$input.path('$.pets')</code> returns an object like this:<pre>
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
</pre><code>$input.path('$.pets').count()</code> would return <code>"3"</code>.

For more information about JSONPath, see JSONPath or JSONPath for Java.</td></tr>
<tr><td><code>$stageVariables.<variable_name></code></td><td><code><variable_name></code> represents a stage variable name.</td></tr>
<tr><td><code>$stageVariables['<variable_name>']</code></td><td><code><variable_name></code> represents any stage variable name.</td></tr>
</table>

Parameter	Description
`${stageVariables['` *`<variable_name>`* `']}`	*`<variable_name>`* represents any stage variable name.
`$util.escapeJavaScript()`	Escapes the characters in a string using JavaScript string rules. **Note** This function will turn any regular single quotes (`'`) into escaped ones (`\'`). However, the escaped single quotes are not valid in JSON. Thus, when the output from this function is used in a JSON property, you must turn any escaped single quotes (`\'`) back to regular single quotes (`'`). This is shown in the following example: `$util.escapeJavaScript( data).replaceAll("\\'","'")`

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td><code>$util.parseJson()</code></td><td>

Takes "stringified" JSON and returns an object representation of the result. You can use the result from this function to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL). For example, if you have the following payload:

```
{"errorMessage":"{\"key1\":\"var1\",\"key2\":{\"arr\":[1,2,3]}}"}
```

and use the following mapping template

```
#set ($errorMessageObj = $util.parseJson($input.path('$.errorMessage')))
{
   "errorMessageObjKey2ArrVal" : $errorMessageObj.key2.arr[0]
}
```

You will get the following output:

```
{
   "errorMessageObjKey2ArrVal" : 1
}
```

</td></tr>
<tr><td><code>$util.urlEncode()</code></td><td>Converts a string into "application/x-www-form-urlencoded" format.</td></tr>
<tr><td><code>$util.urlDecode()</code></td><td>Decodes an "application/x-www-form-urlencoded" string.</td></tr>
<tr><td><code>$util.base64Encode()</code></td><td>Encodes the data into a base64-encoded string.</td></tr>
</table>

Parameter	Description
`$util.base64Decode()`	Decodes the data from a base64-encoded string.

Working with binary media types for WebSocket APIs

API Gateway WebSocket APIs don't currently support binary frames in incoming message payloads. If a client app sends a binary frame, API Gateway rejects it and disconnects the client with code 1003.

There is a workaround for this behavior. If the client sends a text-encoded binary data (e.g., base64) as a text frame, you can set the integration's `contentHandlingStrategy` property to `CONVERT_TO_BINARY` to convert the payload from base64-encoded string to binary.

To return a route response for a binary payload in non-proxy integrations, you can set the integration response's `contentHandlingStrategy` property to `CONVERT_TO_TEXT` to convert the payload from binary to base64-encoded string.

Invoking a WebSocket API

After you've deployed your WebSocket API, client applications can connect to it and send messages to it—and your backend service can send messages to connected client applications:

- You can use `wscat` to connect to your WebSocket API and send messages to it to simulate client behavior. See the section called "Use `wscat` to connect to a WebSocket API and send messages to it".
- You can use the @connections API from your backend service to send a callback message to a connected client, get connection information, or disconnect the client. See the section called "Use `@connections` commands in your backend service".
- A client application can use its own WebSocket library to invoke your WebSocket API.

Use `wscat` to connect to a WebSocket API and send messages to it

The `wscat` utility is a convenient tool for testing a WebSocket API that you have created and deployed in API Gateway. You can install and use `wscat` as follows:

1. Download `wscat` from https://www.npmjs.com/package/wscat.

2. Install `wscat` by running the following command:

   ```
   npm install -g wscat
   ```

3. To connect to your API, run the `wscat` command as shown in the following example. Note that this example assumes that the `Authorization` setting is `NONE`.

   ```
   wscat -c wss://aabbccddee.execute-api.us-east-1.amazonaws.com/test/
   ```

   You need to replace *`aabbccddee`* with the actual API ID, which is displayed in the API Gateway console or returned by the AWS CLI [`create-api`] command.

   In addition, if your API is in a Region other than `us-east-1`, you need to substitute the correct Region.

4. To test your API, enter a message such as the following while connected:

   ```
   {"{jsonpath-expression}":"{route-key}"}
   ```

   where *`{jsonpath-expression}`* is a JSONPath expression and *`{route-key}`* is a route key for the API. For example:

   ```
   {"action":"action1"}
   {"message":"test response body"}
   ```

   For more information about JSONPath, see JSONPath or JSONPath for Java.

5. To disconnect from your API, enter `ctrl-C`.

Use `@connections` commands in your backend service

Your backend service can use the following WebSocket connection HTTP requests to send a callback message to a connected client, get connection information, or disconnect the client.

> ⚠ **Important**
>
> These requests use IAM authorization, so you must sign them with Signature Version 4 (SigV4). To do this, you can use the API Gateway Management API. For more information, see ApiGatewayManagementApi.

In the following command, you need to replace *{api-id}* with the actual API ID, which is displayed in the API Gateway console or returned by the AWS CLI `create-api` command. You must establish the connection before using this command.

To send a callback message to the client, use:

```
POST https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

You can test this request by using `Postman` or by calling `awscurl` as in the following example:

```
awscurl --service execute-api -X POST -d "hello world" https://{prefix}.execute-api.us-
east-1.amazonaws.com/{stage}/@connections/{connection_id}
```

You need to URL-encode the command as in the following example:

```
awscurl --service execute-api -X POST -d "hello world" https://aabbccddee.execute-
api.us-east-1.amazonaws.com/prod/%40connections/R0oXAdfD0kwCH6w%3D
```

To get the latest connection status of the client, use:

```
GET https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

To disconnect the client, use:

```
DELETE https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

You can dynamically build a callback URL by using the `$context` variables in your integration. For example, if you use Lambda proxy integration with a `Node.js` Lambda function, you can build the URL and send a message to a connected client as follows:

```
import {
  ApiGatewayManagementApiClient,
  PostToConnectionCommand,
} from "@aws-sdk/client-apigatewaymanagementapi";

export const handler = async (event) => {
  const domain = event.requestContext.domainName;
  const stage = event.requestContext.stage;
  const connectionId = event.requestContext.connectionId;
  const callbackUrl = `https://${domain}/${stage}`;
  const client = new ApiGatewayManagementApiClient({ endpoint: callbackUrl });

  const requestParams = {
    ConnectionId: connectionId,
    Data: "Hello!",
  };

  const command = new PostToConnectionCommand(requestParams);

  try {
    await client.send(command);
  } catch (error) {
    console.log(error);
  }

  return {
    statusCode: 200,
  };
};
```

When sending a callback message, your Lambda function must have permission to call the API Gateway Management API. You might receive an error that contains `GoneException` if you post a message before the connection is established, or after the client has disconnected.

Publishing WebSocket APIs for customers to invoke

Simply creating and developing an API Gateway API doesn't automatically make it callable by your users. To make it callable, you must deploy your API to a stage. In addition, you might want to customize the URL that your users will use to access your API. You can give it a domain that is consistent with your brand or is more memorable than the default URL for your API.

In this section, you can learn how to deploy your API and customize the URL that you provide to users to access it.

> **Note**
>
> To augment the security of your API Gateway APIs, the `execute-api.{region}.amazonaws.com` domain is registered in the Public Suffix List (PSL). For further

security, we recommend that you use cookies with a `__Host-` prefix if you ever need to set sensitive cookies in the default domain name for your API Gateway APIs. This practice will help to defend your domain against cross-site request forgery attempts (CSRF). For more information see the Set-Cookie page in the Mozilla Developer Network.

Topics

Working with stages for WebSocket APIs

An API stage is a logical reference to a lifecycle state of your API (for example, `dev`, `prod`, `beta`, or `v2`). API stages are identified by their API ID and stage name, and they're included in the URL you use to invoke the API. Each stage is a named reference to a deployment of the API and is made available for client applications to call.

A deployment is a snapshot of your API configuration. After you deploy an API to a stage, it's available for clients to invoke. You must deploy an API for changes to take effect.

Stage variables

Stage variables are key-value pairs that you can define for a stage of a WebSocket API. They act like environment variables and can be used in your API setup.

For example, you can define a stage variable, and then set its value as an HTTP endpoint for an HTTP proxy integration. Later, you can reference the endpoint by using the associated stage variable name. By doing this, you can use the same API setup with a different endpoint at each stage. Similarly, you can use stage variables to specify a different AWS Lambda function integration for each stage of your API.

Note

Stage variables are not intended to be used for sensitive data, such as credentials. To pass sensitive data to integrations, use an AWS Lambda authorizer. You can pass sensitive data to integrations in the output of the Lambda authorizer. To learn more, see the section called “Lambda authorizer response format”.

Examples

To use a stage variable to customize the HTTP integration endpoint, you must first set the name and value of the stage variable (for example, `url`) with a value of `example.com`. Next, set up an HTTP proxy integration. Instead of entering the endpoint's URL, you can tell API Gateway to use the stage variable value, **`http://${stageVariables.url}`**. This value tells API Gateway to substitute your stage variable `${}` at runtime, depending on the stage of your API.

You can reference stage variables in a similar way to specify a Lambda function name or an AWS role ARN.

When specifying a Lambda function name as a stage variable value, you must configure the permissions on the Lambda function manually. You can use the AWS Command Line Interface (AWS CLI) to do this.

```
aws lambda add-permission --function-name arn:aws:lambda:XXXXXX:your-lambda-function-
name --source-arn arn:aws:execute-api:us-east-1:YOUR_ACCOUNT_ID:api_id/*/HTTP_METHOD/
resource --principal apigateway.amazonaws.com --statement-id apigateway-access --action
 lambda:InvokeFunction
```

API Gateway stage variables reference

HTTP integration URIs

You can use a stage variable as part of an HTTP integration URI, as shown in the following examples.

- A full URI without protocol – `http://${stageVariables.<variable_name>}`
- A full domain – `http://${stageVariables.<variable_name>}/resource/operation`
- A subdomain – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- A path – `http://example.com/${stageVariables.<variable_name>}/bar`
- A query string – `http://example.com/foo?q=${stageVariables.<variable_name>}`

Lambda functions

You can use a stage variable in place of a Lambda function name or alias, as shown in the following examples.

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

> **Note**
>
> To use a stage variable for a Lambda function, the function must be in the same account as the API. Stage variables don't support cross-account Lambda functions.

AWS integration credentials

You can use a stage variable as part of an AWS user or role credential ARN, as shown in the following example.

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

Deploy a WebSocket API in API Gateway

After creating your WebSocket API, you must deploy it to make it available for your users to invoke.

To deploy an API, you create an API deployment and associate it with a stage. Each stage is a snapshot of the API and is made available for client apps to call.

> **Important**
>
> Every time you update an API, you must redeploy it. Changes to anything other than stage settings require a redeployment, including modifications to the following resources:
>
> - Routes

- Integrations
- Authorizers

By default you are limited to 10 stages for each API. We recommend that you re-use stages for your deployments.

To call a deployed WebSocket API, the client sends a message to the API's URL. The URL is determined by the API's hostname and stage name.

Note

API Gateway will support payloads up to 128 KB with a maximum frame size of 32 KB. If a message exceeds 32 KB, it must be split into multiple frames, each 32 KB or smaller.

Using the API's default domain name, the URL of (for example) a WebSocket API in a given stage (*{stageName}*) is in the following format:

```
wss://{api-id}.execute-api.{region}.amazonaws.com/{stageName}
```

To make the WebSocket API's URL more user-friendly, you can create a custom domain name (e.g., `api.example.com`) to replace the default host name of the API. The configuration process is the same as for REST APIs. For more information, see the section called "Custom domain names".

Stages enable robust version control of your API. For example, you can deploy an API to a `test` stage and a `prod` stage, and use the `test` stage as a test build and use the `prod` stage as a stable build. After the updates pass the test, you can promote the `test` stage to the `prod` stage. The promotion can be done by redeploying the API to the `prod` stage. For more details about stages, see the section called "Set up a stage".

Topics

- Create a WebSocket API deployment using the AWS CLI
- Create a WebSocket API deployment using the API Gateway console

Create a WebSocket API deployment using the AWS CLI

To use AWS CLI to create a deployment, use the `create-deployment` command as shown in the following example:

```
aws apigatewayv2 --region us-east-1 create-deployment --api-id aabbccddee
```

Example output:

```
{
    "DeploymentId": "fedcba",
    "DeploymentStatus": "DEPLOYED",
    "CreatedDate": "2018-11-15T06:49:09Z"
}
```

The deployed API is not callable until you associate the deployment with a stage. You can create a new stage or reuse a stage that you have previously created.

To create a new stage and associate it with the deployment, use the `create-stage` command as shown in the following example:

```
aws apigatewayv2 --region us-east-1 create-stage --api-id aabbccddee --deployment-id
 fedcba --stage-name test
```

Example output:

```
{
    "StageName": "test",
    "CreatedDate": "2018-11-15T06:50:28Z",
    "DeploymentId": "fedcba",
    "DefaultRouteSettings": {
        "MetricsEnabled": false,
        "ThrottlingBurstLimit": 5000,
        "DataTraceEnabled": false,
        "ThrottlingRateLimit": 10000.0
    },
    "LastUpdatedDate": "2018-11-15T06:50:28Z",
    "StageVariables": {},
    "RouteSettings": {}
}
```

To reuse an existing stage, update the stage's deploymentId property with the newly created deployment ID (*{deployment-id}*) by using the update-stage command.

```
aws apigatewayv2 update-stage --region {region} \
   --api-id {api-id} \
   --stage-name {stage-name} \
   --deployment-id {deployment-id}
```

Create a WebSocket API deployment using the API Gateway console

To use the API Gateway console to create a deployment for a WebSocket API:

1. Sign in to the API Gateway console and choose the API.
2. Choose **Deploy API**.
3. Choose the desired stage from the dropdown list or enter the name of a new stage.

Security policy for WebSocket APIs

API Gateway enforces a security policy of TLS_1_2 for all WebSocket API endpoints.

A *security policy* is a predefined combination of minimum TLS version and cipher suites offered by Amazon API Gateway. The TLS protocol addresses network security problems such as tampering and eavesdropping between a client and server. When your clients establish a TLS handshake to your API through the custom domain, the security policy enforces the TLS version and cipher suite options your clients can choose to use. This security policy accepts TLS 1.2 and TLS 1.3 traffic and rejects TLS 1.0 traffic.

Supported TLS protocols and ciphers for WebSocket APIs

The following table describes the supported TLS protocols and ciphers for WebSocket APIs.

Security policy	**TLS_1_2**
TLS protocols	
TLSv1.3	♦
TLSv1.2	♦

Security policy	TLS_1_2
TLS ciphers	
TLS_AES_128_GCM_SHA256	♦
TLS_AES_256_GCM_SHA384	♦
TLS_CHACHA20_POLY1305_SHA256	♦
ECDHE-ECDSA-AES128-GCM-SHA256	♦
ECDHE-RSA-AES128-GCM-SHA256	♦
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256-GCM-SHA384	♦
ECDHE-RSA-AES256-GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL and RFC cipher names

OpenSSL and IETF RFC 5246, use different names for the same ciphers. For a list of the cipher names, see the section called “OpenSSL and RFC cipher names”.

Information about REST APIs and HTTP APIs

For more information about REST APIs and HTTP APIs, see the section called “Choosing a security policy” and the section called “Security policy for HTTP APIs”.

Setting up custom domain names for WebSocket APIs

Custom domain names are simpler and more intuitive URLs that you can provide to your API users.

After deploying your API, you (and your customers) can invoke the API using the default base URL of the following format:

```
https://api-id.execute-api.region.amazonaws.com/stage
```

where `api-id` is generated by API Gateway, `region` (AWS Region) is specified by you when creating the API, and `stage` is specified by you when deploying the API.

The hostname portion of the URL (that is, *`api-id`*`.execute-api.`*`region`*`.amazonaws.com`) refers to an API endpoint. The default API endpoint can be difficult to recall and not user-friendly.

With custom domain names, you can set up your API's hostname, and choose a base path (for example, `myservice`) to map the alternative URL to your API. For example, a more user-friendly API base URL can become:

```
https://api.example.com/myservice
```

Note

A custom domain name for a WebSocket API can't be mapped to REST APIs or HTTP APIs.
For WebSocket APIs, Regional custom domain names are supported.
For WebSocket APIs, TLS 1.2 is the only supported TLS version.

Register a domain name

You must have a registered internet domain name in order to set up custom domain names for your APIs. If needed, you can register an internet domain using Amazon Route 53 or using a third-party domain registrar of your choice. An API's custom domain name can be the name of a subdomain or the root domain (also known as "zone apex") of a registered internet domain.

After a custom domain name is created in API Gateway, you must create or update your DNS provider's resource record to map to your API endpoint. Without such a mapping, API requests bound for the custom domain name cannot reach API Gateway.

Regional custom domain names

When you create a custom domain name for a Regional API, API Gateway creates a Regional domain name for the API. You must set up a DNS record to map the custom domain name to the Regional domain name. You must also provide a certificate for the custom domain name.

Wildcard custom domain names

With wildcard custom domain names, you can support an almost infinite number of domain names without exceeding the default quota. For example, you could give each of your customers their own domain name, *customername*.api.example.com.

To create a wildcard custom domain name, specify a wildcard (*) as the first subdomain of a custom domain that represents all possible subdomains of a root domain.

For example, the wildcard custom domain name *.example.com results in subdomains such as a.example.com, b.example.com, and c.example.com, which all route to the same domain.

Wildcard custom domain names support distinct configurations from API Gateway's standard custom domain names. For example, in a single AWS account, you can configure *.example.com and a.example.com to behave differently.

You can use the $context.domainName and $context.domainPrefix context variables to determine the domain name that a client used to call your API. To learn more about context variables, see API Gateway mapping template and access logging variable reference.

To create a wildcard custom domain name, you must provide a certificate issued by ACM that has been validated using either the DNS or the email validation method.

> **Note**
>
> You can't create a wildcard custom domain name if a different AWS account has created a custom domain name that conflicts with the wildcard custom domain name. For example, if account A has created a.example.com, then account B can't create the wildcard custom domain name *.example.com.

If account A and account B share an owner, you can contact the AWS Support Center to request an exception.

Certificates for custom domain names

⚠ **Important**

You specify the certificate for your custom domain name. If your application uses certificate pinning, sometimes known as SSL pinning, to pin an ACM certificate, the application might not be able to connect to your domain after AWS renews the certificate. For more information, see Certificate pinning problems in the *AWS Certificate Manager User Guide*.

To provide a certificate for a custom domain name in a Region where ACM is supported, you must request a certificate from ACM. To provide a certificate for a Regional custom domain name in a Region where ACM is not supported, you must import a certificate to API Gateway in that Region.

To import an SSL/TLS certificate, you must provide the PEM-formatted SSL/TLS certificate body, its private key, and the certificate chain for the custom domain name. Each certificate stored in ACM is identified by its ARN. To use an AWS managed certificate for a domain name, you simply reference its ARN.

ACM makes it straightforward to set up and use a custom domain name for an API. You create a certificate for the given domain name (or import a certificate), set up the domain name in API Gateway with the ARN of the certificate provided by ACM, and map a base path under the custom domain name to a deployed stage of the API. With certificates issued by ACM, you do not have to worry about exposing any sensitive certificate details, such as the private key.

Set up a custom domain name

For details on setting up a custom domain name, see Getting certificates ready in AWS Certificate Manager and Setting up a regional custom domain name in API Gateway.

Working with API mappings for WebSocket APIs

You use API mappings to connect API stages to a custom domain name. After you create a domain name and configure DNS records, you use API mappings to send traffic to your APIs through your custom domain name.

An API mapping specifies an API, a stage, and optionally a path to use for the mapping. For example, you can map the `production` stage of an API to `wss://api.example.com/orders`.

Before you create an API mapping, you must have an API, a stage, and a custom domain name. To learn more about creating a custom domain name, see the section called "Setting up a regional custom domain name".

Restrictions

- In an API mapping, the custom domain name and mapped APIs must be in the same AWS account.
- API mappings must contain only letters, numbers, and the following characters: `$-_.+!*'()`.
- The maximum length for the path in an API mapping is 300 characters.
- You can't map WebSocket APIs to the same custom domain name as an HTTP API or REST API.

Create an API mapping

To create an API mapping, you must first create a custom domain name, API, and stage. For information about creating a custom domain name, see the section called "Setting up a regional custom domain name".

AWS Management Console

To create an API mapping

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose **Custom domain names**.
3. Select a custom domain name that you've already created.
4. Choose **API mappings**.
5. Choose **Configure API mappings**.
6. Choose **Add new mapping**.
7. Enter an **API**, a **Stage**, and optionally a **Path**.
8. Choose **Save**.

AWS CLI

The following AWS CLI command creates an API mapping. In this example, API Gateway sends requests to `api.example.com/v1` to the specified API and stage.

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1 \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

The following AWS CloudFormation example creates an API mapping.

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'v1'
    ApiId: !Ref MyApi
    Stage: !Ref MyStage
```

Disabling the default endpoint for a WebSocket API

By default, clients can invoke your API by using the `execute-api` endpoint that API Gateway generates for your API. To ensure that clients can access your API only by using a custom domain name, disable the default `execute-api` endpoint.

Note

When you disable the default endpoint, it affects all stages of an API.

The following AWS CLI command disables the default endpoint for an WebSocket API.

```
aws apigatewayv2 update-api \
    --api-id abcdef123 \
    --disable-execute-api-endpoint
```

After you disable the default endpoint, you must deploy your API for the change to take effect.

The following AWS CLI command creates a deployment.

```
aws apigatewayv2 create-deployment \
    --api-id abcdef123 \
    --stage-name dev
```

Protecting your WebSocket API

You can configure throttling for your APIs to help protect them from being overwhelmed by too many requests. Throttles are applied on a best-effort basis and should be thought of as targets rather than guaranteed request ceilings.

API Gateway throttles requests to your API using the token bucket algorithm, where a token counts for a request. Specifically, API Gateway examines the rate and a burst of request submissions against all APIs in your account, per Region. In the token bucket algorithm, a burst can allow pre-defined overrun of those limits, but other factors can also cause limits to be overrun in some cases.

When request submissions exceed the steady-state request rate and burst limits, API Gateway begins to throttle requests. Clients may receive `429 Too Many Requests` error responses at this point. Upon catching such exceptions, the client can resubmit the failed requests in a way that is rate limiting.

As an API developer, you can set the target limits for individual API stages or routes to improve overall performance across all APIs in your account.

Account-level throttling per Region

By default, API Gateway limits the steady-state requests per second (RPS) across all APIs within an AWS account, per Region. It also limits the burst (that is, the maximum bucket size) across all APIs within an AWS account, per Region. In API Gateway, the burst limit represents the target maximum number of concurrent request submissions that API Gateway will fulfill before returning `429 Too Many Requests` error responses. For more information on throttling quotas, see *Quotas and important notes*.

Per-account limits are applied to all APIs in an account in a specified Region. The account-level rate limit can be increased upon request - higher limits are possible with APIs that have shorter timeouts and smaller payloads. To request an increase of account-level throttling limits per Region, contact the AWS Support Center. For more information, see *Quotas and important notes*. Note that these limits can't be higher than the AWS throttling limits.

Route-level throttling

You can set route-level throttling to override the account-level request throttling limits for a specific stage or for individual routes in your API. The default route throttling limits can't exceed account-level rate limits.

You can configure route-level throttling by using the AWS CLI. The following command configures custom throttling for the specified stage and route of an API.

```
aws apigatewayv2 update-stage \
    --api-id a1b2c3d4 \
    --stage-name dev \
    --route-settings '{"messages":
{"ThrottlingBurstLimit":100,"ThrottlingRateLimit":2000}}'
```

Monitoring WebSocket APIs

You can use CloudWatch metrics and CloudWatch Logs to monitor WebSocket APIs. By combining logs and metrics, you can log errors and monitor your API's performance.

> **Note**
>
> API Gateway might not generate logs and metrics in the following cases:
>
> - 413 Request Entity Too Large errors
> - Excessive 429 Too Many Requests errors
> - 400 series errors from requests sent to a custom domain that has no API mapping
> - 500 series errors caused by internal failures

Topics

Monitoring WebSocket API execution with CloudWatch metrics

You can use Amazon CloudWatch metrics to monitor WebSocket APIs. The configuration is similar to that used for REST APIs. For more information, see Monitoring REST API execution with Amazon CloudWatch metrics.

The following metrics are supported for WebSocket APIs:

Metric	Description
ConnectCount	The number of messages sent to the `$connect` route integration.
MessageCount	The number of messages sent to the WebSocket API, either from or to the client.
IntegrationError	The number of requests that return a 4XX/5XX response from the integration.
ClientError	The number of requests that have a 4XX response returned by API Gateway before the integration is invoked.
ExecutionError	Errors that occurred when calling the integration.
IntegrationLatency	The time difference between API Gateway sending the request to the integration and API Gateway receiving the response from the integrati on. Suppressed for callbacks and mock integrations.

You can use the dimensions in the following table to filter API Gateway metrics.

Dimension	Description
ApiId	Filters API Gateway metrics for an API with the specified API ID.
ApiId, Stage	Filters API Gateway metrics for an API stage with the specified API ID and stage ID.
ApiId, Method, Resource, Stage	Filters API Gateway metrics for an API method with the specified API ID, stage ID, resource path, and route ID. API Gateway will not send these metrics unless you have explicitly enabled detailed CloudWatch metrics. You can do this by calling the UpdateStage action of the API Gateway V2 REST API to update the `detailedMetricsEnabled` property to `true`. Alternatively, you can call the update-stage AWS CLI command to update the `DetailedMetricsEnabled` property to `true`. Enabling such metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing.

Configuring logging for a WebSocket API

You can enable logging to write logs to CloudWatch Logs. There are two types of API logging in CloudWatch: execution logging and access logging. In execution logging, API Gateway manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams any caller's requests and responses.

In access logging, you, as an API developer, want to log who has accessed your API and how the caller accessed the API. You can create your own log group or choose an existing log group that could be managed by API Gateway. To specify the access details, you select `$context` variables (expressed in a format of your choosing) and choose a log group as the destination.

For instructions on how to set up CloudWatch logging, see the section called "Set up CloudWatch API logging using the API Gateway console".

When you specify the **Log Format**, you can choose which context variables to log. The following variables are supported.

Parameter	Description
`$context.apiId`	The identifier API Gateway assigns to your API.
`$context.authorize.error`	The authorization error message.
`$context.authorize.latency`	The authorization latency in ms.
`$context.authorize.status`	The status code returned from an authorization attempt.
`$context.authorizer.error`	The error message returned from an authorizer.
`$context.authorizer.integrationLatency`	The Lambda authorizer latency in ms.
`$context.authorizer.integrationStatus`	The status code returned from a Lambda authorizer.
`$context.authorizer.latency`	The authorizer latency in ms.

Parameter	Description
`$context.authorizer.requestId`	The AWS endpoint's request ID.
`$context.authorizer.status`	The status code returned from an authorizer.
`$context.authorizer.principalId`	The principal user identification that is associated with the token sent by the client and returned from an API Gateway Lambda authorizer Lambda function. (A Lambda authorizer was formerly known as a custom authorizer.)

`$context.authorizer.`*`property`*	The stringified value of the specified key-value pair of the `context` map returned from an API Gateway Lambda authorizer function. For example, if the authorizer returns the following `context` map:
	`"context" : {`
	`"key":`
	`"value",`
	`"numKey":`
	`1,`
	`"boolKey":`
	`true`
	`}`
	calling `$context.authorizer.key` returns the `"value"` string, calling `$context.authorizer.numKey` returns the `"1"` string, and calling `$context.authorizer.boolKey` returns the `"true"` string.

`$context.authenticate.error`	The error message returned from an authentication attempt.
`$context.authenticate.latency`	The authentication latency in ms.

Parameter	Description
`$context.authenticate.status`	The status code returned from an authentication attempt.
`$context.connectedAt`	The Epoch-formatted connection time.
`$context.connectionId`	A unique ID for the connection that can be used to make a callback to the client.
`$context.domainName`	A domain name for the WebSocket API. This can be used to make a callback to the client (instead of a hardcoded value).
`$context.error.message`	A string that contains an API Gateway error message.
`$context.error.messageString`	The quoted value of `$context.error.message` , namely `"$context.error.message"` .
`$context.error.responseType`	The error response type.
`$context.error.validationErrorString`	A string that contains a detailed validation error message.
`$context.eventType`	The event type: `CONNECT`, `MESSAGE`, or `DISCONNECT` .
`$context.extendedRequestId`	Equivalent to `$context.requestId` .
`$context.identity.accountId`	The AWS account ID associated with the request.
`$context.identity.apiKey`	The API owner key associated with key-enabled API request.
`$context.identity.apiKeyId`	The API key ID associated with the key-enabled API request

Parameter	Description
`$context.identity.caller`	The principal identifier of the caller that signed the request. Supported for routes that use IAM authorization.
`$context.identity.cognitoAuthenticationProvider`	A comma-separated list of the Amazon Cognito authentication providers used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For example, for an identity from an Amazon Cognito user pool, `cognito-idp. region.amazonaws.com/ user_pool_id ,cognito-idp. region.amazonaws.com/ user_pool_id :CognitoSignIn: token subject claim` For information, see [Using Federated Identities](#) in the *Amazon Cognito Developer Guide*.
`$context.identity.cognitoAuthenticationType`	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials. Possible values include `authenticated` for authenticated identities and `unauthenticated` for unauthenticated identities.
`$context.identity.cognitoIdentityId`	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.

Parameter	Description
`$context.identity.cognitoIdentityPoolId`	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.principalOrgId`	The AWS organization ID. Supported for routes that use IAM authorization.
`$context.identity.sourceIp`	The source IP address of the TCP connection making the request to API Gateway.
`$context.identity.user`	The principal identifier of the user that will be authorized against resource access. Supported for routes that use IAM authorization.
`$context.identity.userAgent`	The user agent of the API caller.
`$context.identity.userArn`	The Amazon Resource Name (ARN) of the effective user identified after authentication.
`$context.integration.error`	The error message returned from an integration.
`$context.integration.integrationStatus`	For Lambda proxy integration, the status code returned from AWS Lambda, not from the backend Lambda function code.
`$context.integration.latency`	The integration latency in ms. Equivalent to `$context.integrationLatency` .
`$context.integration.requestId`	The AWS endpoint's request ID. Equivalent to `$context.awsEndpointRequestId` .

Parameter	Description
`$context.integration.status`	The status code returned from an integration. For Lambda proxy integrations, this is the status code that your Lambda function code returns. Equivalent to `$context.integrationStatus` .
`$context.integrationLatency`	The integration latency in ms, available for access logging only.
`$context.messageId`	A unique server-side ID for a message. Available only when the `$context.eventType` is `MESSAGE`.
`$context.requestId`	Same as `$context.extendedRequestId` .
`$context.requestTime`	The CLF-formatted request time (`dd/MMM/yyyy:HH:mm:ss +-hhmm`).
`$context.requestTimeEpoch`	The Epoch-formatted request time, in milliseconds.
`$context.routeKey`	The selected route key.
`$context.stage`	The deployment stage of the API call (for example, beta or prod).
`$context.status`	The response status.
`$context.waf.error`	The error message returned from AWS WAF.
`$context.waf.latency`	The AWS WAF latency in ms.
`$context.waf.status`	The status code returned from AWS WAF.

Examples of some commonly used access log formats are shown in the API Gateway console and are listed as follows.

- CLF (Common Log Format):

```
$context.identity.sourceIp $context.identity.caller \
$context.identity.user [$context.requestTime] "$context.eventType $context.routeKey
 $context.connectionId" \
$context.status $context.requestId
```

  The continuation characters (\) are meant as a visual aid. The log format must be a single line. You can add a newline character (\n) at the end of the log format to include a newline at the end of each log entry.

- JSON:

```
{
"requestId":"$context.requestId", \
"ip": "$context.identity.sourceIp", \
"caller":"$context.identity.caller", \
"user":"$context.identity.user", \
"requestTime":"$context.requestTime", \
"eventType":"$context.eventType", \
"routeKey":"$context.routeKey", \
"status":"$context.status", \
"connectionId":"$context.connectionId"
}
```

  The continuation characters (\) are meant as a visual aid. The log format must be a single line. You can add a newline character (\n) at the end of the log format to include a newline at the end of each log entry.

- XML:

```
<request id="$context.requestId"> \
 <ip>$context.identity.sourceIp</ip> \
 <caller>$context.identity.caller</caller> \
 <user>$context.identity.user</user> \
 <requestTime>$context.requestTime</requestTime> \
 <eventType>$context.eventType</eventType> \
 <routeKey>$context.routeKey</routeKey> \
 <status>$context.status</status> \
 <connectionId>$context.connectionId</connectionId> \
</request>
```

The continuation characters (\) are meant as a visual aid. The log format must be a single line. You can add a newline character (\n) at the end of the log format to include a newline at the end of each log entry.

- CSV (comma-separated values):

```
$context.identity.sourceIp,$context.identity.caller, \
$context.identity.user,$context.requestTime,$context.eventType, \
$context.routeKey,$context.connectionId,$context.status, \
$context.requestId
```

The continuation characters (\) are meant as a visual aid. The log format must be a single line. You can add a newline character (\n) at the end of the log format to include a newline at the end of each log entry.

API Gateway Amazon Resource Name (ARN) reference

The following tables list the Amazon Resource Names (ARNs) for API Gateway resources. To learn more about using ARNs in AWS Identity and Access Management policies, see How Amazon API Gateway works with IAM and Control access to an API with IAM permissions.

HTTP API and WebSocket API resources

Resource	ARN
AccessLogSettings	arn:*partition*:apigateway:*region*::/apis/*api-id*/stages/*stage-name*/accesslogsettings
Api	arn:*partition*:apigateway:*region*::/apis/*api-id*
Apis	arn:*partition*:apigateway:*region*::/apis
ApiMapping	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*/apimappings/*id*
ApiMappings	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*/apimappings
Authorizer	arn:*partition*:apigateway:*region*::/apis/*api-id*/authorizers/*id*
Authorizers	arn:*partition*:apigateway:*region*::/apis/*api-id*/authorizers

Resource	ARN
Cors	arn:*partition*:apigateway:*region*::/apis/*api-id*/cors
Deployment	arn:*partition*:apigateway:*region*::/apis/*api-id*/deployments/*id*
Deployments	arn:*partition*:apigateway:*region*::/apis/*api-id*/deployments
DomainName	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*
DomainNames	arn:*partition*:apigateway:*region*::/domainnames
ExportedAPI	arn:*partition*:apigateway:*region*::/apis/*api-id*/exports/*specification*
Integration	arn:*partition*:apigateway:*region*::/apis/*api-id*/integrations/*integration-id*
Integrations	arn:*partition*:apigateway:*region*::/apis/*api-id*/integrations
IntegrationResponse	arn:*partition*:apigateway:*region*::/apis/*api-id*/integrationresponses/*integration-response*

Resource	ARN
IntegrationResponses	`arn:partition :apigateway: region::/apis/api-id/integrationresponses`
Model	`arn:partition :apigateway: region::/apis/api-id/models/id`
Models	`arn:partition :apigateway: region::/apis/api-id/models`
ModelTemplate	`arn:partition :apigateway: region::/apis/api-id/models/id/template`
Route	`arn:partition :apigateway: region::/apis/api-id/routes/id`
Routes	`arn:partition :apigateway: region::/apis/api-id/routes`
RouteRequestParameter	`arn:partition :apigateway: region::/apis/api-id/routes/id/requestparameters/ key`
RouteResponse	`arn:partition :apigateway: region::/apis/api-id/routes/id/routeresponses/ id`
RouteResponses	`arn:partition :apigateway: region::/apis/api-id/routes/id/routeresponses`
RouteSettings	`arn:partition :apigateway: region::/apis/api-id/stages/stage-name /routesettings/ route-key`

Resource	ARN
Stage	arn:*partition* :apigateway: *region*::/apis/*api-id*/stages/*stage-name*
Stages	arn:*partition* :apigateway: *region*::/apis/*api-id*/stages
VpcLink	arn:*partition* :apigateway: *region*::/vpclinks/ *vpclink-id*
VpcLinks	arn:*partition* :apigateway: *region*::/vpclinks

REST API resources

Resource	ARN
Account	arn:*partition* :apigateway: *region*::/account
ApiKey	arn:*partition* :apigateway: *region*::/apikeys/ *id*
ApiKeys	arn:*partition* :apigateway: *region*::/apikeys
Authorizer	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/authorizers/ *id*
Authorizers	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/authorizers

Resource	ARN
BasePathMapping	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*/basepathmappings/*basepath*
BasePathMappings	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*/basepathmappings
ClientCertificate	arn:*partition*:apigateway:*region*::/clientcertificates/*id*
ClientCertificates	arn:*partition*:apigateway:*region*::/clientcertificates
Deployment	arn:*partition*:apigateway:*region*::/restapis/*api-id*/deployments/*id*
Deployments	arn:*partition*:apigateway:*region*::/restapis/*api-id*/deployments
DocumentationPart	arn:*partition*:apigateway:*region*::/restapis/*api-id*/documentation/parts/*id*
DocumentationParts	arn:*partition*:apigateway:*region*::/restapis/*api-id*/documentation/parts
DocumentationVersion	arn:*partition*:apigateway:*region*::/restapis/*api-id*/documentation/versions/*version*

Resource	ARN
DocumentationVersions	arn:*partition*:apigateway:*region*::/restapis/*api-id*/documentation/versions
DomainName	arn:*partition*:apigateway:*region*::/domainnames/*domain-name*
DomainNames	arn:*partition*:apigateway:*region*::/domainnames
GatewayResponse	arn:*partition*:apigateway:*region*::/restapis/*api-id*/gatewayresponses/*response-type*
GatewayResponses	arn:*partition*:apigateway:*region*::/restapis/*api-id*/gatewayresponses
Integration	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/*http-method*/integration
IntegrationResponse	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/*http-method*/integration/responses/*status-code*
Method	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/*http-method*

Resource	ARN
MethodResponse	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/*http-method*/responses/*status-code*
Model	arn:*partition*:apigateway:*region*::/restapis/*api-id*/models/*model-name*
Models	arn:*partition*:apigateway:*region*::/restapis/*api-id*/models
RequestValidator	arn:*partition*:apigateway:*region*::/restapis/*api-id*/requestvalidators/*id*
RequestValidators	arn:*partition*:apigateway:*region*::/restapis/*api-id*/requestvalidators
Resource	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources/*id*
Resources	arn:*partition*:apigateway:*region*::/restapis/*api-id*/resources
RestApi	arn:*partition*:apigateway:*region*::/restapis/*api-id*
RestApis	arn:*partition*:apigateway:*region*::/restapis

Resource	ARN
Stage	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/stages/*stage-name*
Stages	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/stages
Tags	arn:*partition* :apigateway: *region*::/tags/*url-encoded-resource-arn*
Template	arn:*partition* :apigateway: *region*::/restapis/models/ *model-name* /template
UsagePlan	arn:*partition* :apigateway: *region*::/usageplans/ *usageplan-id*
UsagePlans	arn:*partition* :apigateway: *region*::/usageplans
UsagePlanKey	arn:*partition* :apigateway: *region*::/usageplans/ *usageplan-id* /keys/*id*
UsagePlanKeys	arn:*partition* :apigateway: *region*::/usageplans/ *usageplan-id* /keys
VpcLink	arn:*partition* :apigateway: *region*::/vpclinks/ *vpclink-id*
VpcLinks	arn:*partition* :apigateway: *region*::/vpclinks

`execute-api` (HTTP APIs, WebSocket APIs, and REST APIs)

Resource	ARN
WebSocket API endpoint	`arn:`*`partition`*` :execute-api: `*`region`*`:`*`account-id`*` :`*`api-id`*`/`*`stage`*`/`*`route-key`*
HTTP API and REST API endpoint *	`arn:`*`partition`*` :execute-api: `*`region`*`:`*`account-id`*` :`*`api-id`*`/`*`stage`*`/`*`http-method`*` /`*`resource-path`*
Lambda authorizer **	`arn:`*`partition`*` :execute-api: `*`region`*`:`*`account-id`*` :`*`api-id`*`/ authorizers/ `*`authorizer-id`*

* The ARN for the `$default` route endpoint for HTTP APIs is `arn:`*`partition`*`:execute-api:`*`region`*`:`*`account-id`*`:`*`api-id`*`/`*`*`*`/$default`.

** This ARN is applicable only when setting the `SourceArn` condition in the resource policy for a Lambda authorizer function. For an example, see the section called "Create a Lambda authorizer".

Working with API Gateway extensions to OpenAPI

The API Gateway extensions support the AWS-specific authorization and API Gateway-specific API integrations for REST APIs and HTTP APIs. In this section, we describe the API Gateway extensions to the OpenAPI specification.

> **Tip**
>
> To understand how the API Gateway extensions are used in an application, you can use the API Gateway console to create a REST API or HTTP API and export it to an OpenAPI definition file. For more information on how to export an API, see Export a REST API from API Gateway and Exporting an HTTP API from API Gateway.

Topics

- x-amazon-apigateway-integration.requestParameters object
- x-amazon-apigateway-integration.responses object
- x-amazon-apigateway-integration.response object
- x-amazon-apigateway-integration.responseTemplates object
- x-amazon-apigateway-integration.responseParameters object
- x-amazon-apigateway-integration.tlsConfig object
- x-amazon-apigateway-minimum-compression-size
- x-amazon-apigateway-policy
- x-amazon-apigateway-request-validator property
- x-amazon-apigateway-request-validators object
- x-amazon-apigateway-request-validators.requestValidator object
- x-amazon-apigateway-tag-value property

x-amazon-apigateway-any-method object

Specifies the OpenAPI Operation Object for the API Gateway catch-all ANY method in an OpenAPI Path Item Object. This object can exist alongside other Operation objects and will catch any HTTP method that wasn't explicitly declared.

The following table lists the properties extended by API Gateway. For the other OpenAPI Operation properties, see the OpenAPI specification.

Properties

Property name	Type	Description
`isDefaultRoute`	`Boolean`	Specifies whether a route is the `$default` route. Supported only for HTTP APIs. To learn more, see Working with routes for HTTP APIs.
`x-amazon-apigateway-integration`	x-amazon-apigateway-integration object	Specifies the integration of the method with the backend.

Property name	Type	Description
		This is an extended property of the OpenAPI Operation object. The integration can be of type `AWS`, `AWS_PROXY`, `HTTP`, `HTTP_PROXY`, or `MOCK`.

x-amazon-apigateway-any-method examples

The following example integrates the `ANY` method on a proxy resource, `{proxy+}`, with a Lambda function, `TestSimpleProxy`.

```
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:TestSimpleProxy/invocations",
          "httpMethod": "POST",
          "type": "aws_proxy"
        }
```

The following example creates a `$default` route for an HTTP API that integrates with a Lambda function, `HelloWorld`.

```
"/$default": {
    "x-amazon-apigateway-any-method": {
      "isDefaultRoute": true,
      "x-amazon-apigateway-integration": {
        "type": "AWS_PROXY",
        "httpMethod": "POST",
        "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations",
        "timeoutInMillis": 1000,
        "connectionType": "INTERNET",
        "payloadFormatVersion": 1.0
      }
   }
}
```

x-amazon-apigateway-cors object

Specifies the cross-origin resource sharing (CORS) configuration for an HTTP API. The extension applies to the root-level OpenAPI structure. To learn more, see Configuring CORS for an HTTP API.

Properties

Property name	Type	Description
allowOrigins	Array	Specifies the allowed origins.
allowCredentials	Boolean	Specifies whether credentials are included in the CORS request.
exposeHeaders	Array	Specifies the headers that are exposed.
maxAge	Integer	Specifies the number of seconds that the browser should cache preflight request results.
allowMethods	Array	Specifies the allowed HTTP methods.
allowHeaders	Array	Specifies the allowed headers.

x-amazon-apigateway-cors example

The following is an example CORS configuration for an HTTP API.

```
"x-amazon-apigateway-cors": {
    "allowOrigins": [
      "https://www.example.com"
    ],
    "allowCredentials": true,
    "exposeHeaders": [
      "x-apigateway-header",
      "x-amz-date",
      "content-type"
    ],
    "maxAge": 3600,
    "allowMethods": [
      "GET",
      "OPTIONS",
      "POST"
    ],
    "allowHeaders": [
      "x-apigateway-header",
      "x-amz-date",
      "content-type"
    ]
}
```

x-amazon-apigateway-api-key-source property

Specify the source to receive an API key to throttle API methods that require a key. This API-level property is a `String` type.

Specify the source of the API key for requests. Valid values are:

- `HEADER` for receiving the API key from the `X-API-Key` header of a request.
- `AUTHORIZER` for receiving the API key from the `UsageIdentifierKey` from a Lambda authorizer (formerly known as a custom authorizer).

x-amazon-apigateway-api-key-source example

The following example sets the `X-API-Key` header as the API key source.

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "title" : "Test1"
   },
  "schemes" : [ "https" ],
  "basePath" : "/import",
  "x-amazon-apigateway-api-key-source" : "HEADER",
   .
   .
   .
}
```

OpenAPI 3.0.1

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "Test1"
  },
  "servers" : [ {
    "url" : "/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "import"
      }
    }
  } ],
  "x-amazon-apigateway-api-key-source" : "HEADER",
   .
   .
   .
}
```

x-amazon-apigateway-auth object

Defines an authorization type to be applied for authorization of method invocations in API Gateway.

Properties

Property name	Type	Description
`type`	`string`	Specifies the authorization type. Specify `"NONE"` for open access. Specify `"AWS_IAM"` to use IAM permissions. Values are case insensitive.

x-amazon-apigateway-auth example

The following example sets the authorization type for an API method.

OpenAPI 3.0.1

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "openapi3",
    "version": "1.0"
  },
  "paths": {
    "/protected-by-iam": {
      "get": {
        "x-amazon-apigateway-auth": {
          "type": "AWS_IAM"
        }
      }
    }
  }
}
```

x-amazon-apigateway-authorizer object

Defines a Lambda authorizer, Amazon Cognito user pool, or JWT authorizer to be applied for authorization of method invocations in API Gateway. This extension applies to the security definition in OpenAPI 2 and OpenAPI 3.

Properties

Property name	Type	Description
`type`	`string`	The type of the authorizer. This is a required property. For REST APIs, specify `token` for an authorizer with the caller identity embedded in an authorization token. Specify `request` for an authorizer with the caller identity contained in request parameters. Specify `cognito_user_pools` for an authorizer that uses an Amazon Cognito user pool to control access to your API. For HTTP APIs, specify `request` for a Lambda authorizer with the caller identity contained in request parameters. Specify `jwt` for a JWT authorizer.
`authorizerUri`	`string`	The Uniform Resource Identifier (URI) of the authorizer Lambda function. The syntax is as follows:

Property name	Type	Description
		`"arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:account-id:function:auth_function_name/invocations"`
`authorizerCredentials`	`string`	The credentials required for invoking the authorizer, if any, in the form of an ARN of an IAM execution role. For example, "arn:aws:iam::*account-id*:*IAM_role*".
`authorizerPayloadFormatVersion`	`string`	For HTTP APIs, specifies the format of the data that API Gateway sends to a Lambda authorizer, and how API Gateway interprets the response from Lambda. To learn more, see the section called “Payload format version”.

Property name	Type	Description
`enableSimpleResponses`	`Boolean`	For HTTP APIs, specifies whether a `request` authorizer returns a Boolean value or an IAM policy. Supported only for authorizers with an `authorizerPayloadFormatVersion` of `2.0`. If enabled, the Lambda authorizer function returns a Boolean value. To learn more, see the section called "Lambda function response for format 2.0".
`identitySource`	`string`	A comma-separated list of mapping expressions of the request parameters as the identity source. Applicable for the authorizer of the `request` and `jwt` type only.
`jwtConfiguration`	`Object`	Specifies the issuer and audiences for a JWT authorizer. To learn more, see JWTConfiguration in the API Gateway Version 2 API Reference. Supported only for HTTP APIs.
`identityValidationExpression`	`string`	A regular expression for validating the token as the incoming identity. For example, "^x-[a-z]+". Supported only for REST APIs.

Property name	Type	Description
`authorizerResultTtlInSeconds`	`string`	The number of seconds during which authorizer result is cached.
`providerARNs`	An array of `string`	A list of the Amazon Cognito user pool ARNs for the `COGNITO_USER_POOLS` .

x-amazon-apigateway-authorizer examples for REST APIs

The following OpenAPI security definitions example specifies a Lambda authorizer of the "token" type and named `test-authorizer`.

```
  "securityDefinitions" : {
    "test-authorizer" : {
      "type" : "apiKey",                         // Required and the value must be
 "apiKey" for an API Gateway API.
      "name" : "Authorization",                  // The name of the header containing
 the authorization token.
      "in" : "header",                           // Required and the value must be
 "header" for an API Gateway API.
      "x-amazon-apigateway-authtype" : "oauth2", // Specifies the authorization
 mechanism for the client.
      "x-amazon-apigateway-authorizer" : {       // An API Gateway Lambda authorizer
 definition
        "type" : "token",                        // Required property and the value
 must "token"
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:account-id:function:function-name/invocations",
        "authorizerCredentials" : "arn:aws:iam::account-id:role",
        "identityValidationExpression" : "^x-[a-z]+",
        "authorizerResultTtlInSeconds" : 60
      }
    }
  }
```

The following OpenAPI operation object snippet sets the `GET /http` to use the preceding Lambda authorizer.

```
   "/http" : {
      "get" : {
        "responses" : { },
        "security" : [ {
          "test-authorizer" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "httpMethod" : "GET",
          "uri" : "http://api.example.com"
        }
      }
    }
```

The following OpenAPI security definitions example specifies a Lambda authorizer of the "request" type, with a single header parameter (`auth`) as the identity source. The `securityDefinitions` is named `request_authorizer_single_header`.

```
"securityDefinitions": {
    "request_authorizer_single_header" : {
      "type" : "apiKey",
      "name" : "auth",               // The name of a single header or query parameter
 as the identity source.
      "in" : "header",               // The location of the single identity source
 request parameter. The valid value is "header" or "query"
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "method.request.header.auth",   // Request parameter mapping
 expression of the identity source. In this example, it is the 'auth' header.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

The following OpenAPI security definitions example specifies a Lambda authorizer of the "request" type, with one header (`HeaderAuth1`) and one query string parameter `QueryString1` as the identity sources.

```
"securityDefinitions": {
    "request_authorizer_header_query" : {
      "type" : "apiKey",
      "name" : "Unused",             // Must be "Unused" for multiple identity sources
 or non header or query type of request parameters.
      "in" : "header",               // Must be "header" for multiple identity sources
 or non header or query type of request parameters.
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "method.request.header.HeaderAuth1,
 method.request.querystring.QueryString1",   // Request parameter mapping expressions
 of the identity sources.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

The following OpenAPI security definitions example specifies an API Gateway Lambda authorizer of the "request" type, with a single stage variable (`stage`) as the identity source.

```
"securityDefinitions": {
    "request_authorizer_single_stagevar" : {
      "type" : "apiKey",
      "name" : "Unused",            // Must be "Unused", for multiple identity sources
 or non header or query type of request parameters.
      "in" : "header",              // Must be "header", for multiple identity sources
 or non header or query type of request parameters.
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "stageVariables.stage",   // Request parameter mapping
 expression of the identity source. In this example, it is the stage variable.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

The following OpenAPI security definition example specifies an Amazon Cognito user pool as an authorizer.

```
 "securityDefinitions": {
    "cognito-pool": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "cognito_user_pools",
      "x-amazon-apigateway-authorizer": {
        "type": "cognito_user_pools",
        "providerARNs": [
          "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-east-1_ABC123"
        ]
      }
    }
```

The following OpenAPI operation object snippet sets the `GET /http` to use the preceding Amazon Cognito user pool as an authorizer, with no custom scopes.

```

   "/http" : {
      "get" : {
        "responses" : { },
        "security" : [ {
          "cognito-pool" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "httpMethod" : "GET",
          "uri" : "http://api.example.com"
        }
      }
    }
```

x-amazon-apigateway-authorizer examples for HTTP APIs

The following OpenAPI 3.0 example creates a JWT authorizer for an HTTP API that uses Amazon Cognito as an identity provider, with the `Authorization` header as an identity source.

```
"securitySchemes": {
  "jwt-authorizer-oauth": {
    "type": "oauth2",
     "x-amazon-apigateway-authorizer": {
       "type": "jwt",
       "jwtConfiguration": {
          "issuer": "https://cognito-idp.region.amazonaws.com/userPoolId",
          "audience": [
            "audience1",
            "audience2"
          ]
        },
        "identitySource": "$request.header.Authorization"
    }
  }
}
```

The following OpenAPI 3.0 example produces the same JWT authorizer as the previous example. However, this example uses the OpenAPI `openIdConnectUrl` property to automatically detect the issuer. The `openIdConnectUrl` must be fully formed.

```
"securitySchemes": {
  "jwt-authorizer-autofind": {
    "type": "openIdConnect",
    "openIdConnectUrl": "https://cognito-idp.region.amazonaws.com/userPoolId/.well-
known/openid-configuration",
    "x-amazon-apigateway-authorizer": {
      "type": "jwt",
      "jwtConfiguration": {
        "audience": [
          "audience1",
          "audience2"
        ]
      },
      "identitySource": "$request.header.Authorization"
    }
  }
}
```

The following example creates a Lambda authorizer for an HTTP API. This example authorizer uses the `Authorization` header as its identity source. The authorizer uses the `2.0` payload format version, and returns Boolean value, because `enableSimpleResponses` is set to `true`.

```
"securitySchemes" : {
  "lambda-authorizer" : {
    "type" : "apiKey",
    "name" : "Authorization",
    "in" : "header",
    "x-amazon-apigateway-authorizer" : {
      "type" : "request",
      "identitySource" : "$request.header.Authorization",
      "authorizerUri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:function-name/invocations",
      "authorizerPayloadFormatVersion" : "2.0",
      "authorizerResultTtlInSeconds" : 300,
      "enableSimpleResponses" : true
    }
  }
}
```

x-amazon-apigateway-authtype property

For REST APIs, this extension can be used to define a custom type of a Lambda authorizer. In this case, the value is free-form. For example, an API may have multiple Lambda authorizers that use different internal schemes. You can use this extension to identify the internal scheme of a Lambda authorizer.

More commonly, in HTTP APIs and REST APIs, it can also be used as a way to define IAM authorization across several operations that share the same security scheme. In this case, the term `awsSigv4` is a reserved term, along with any term prefixed by `aws`.

This extension applies to the `apiKey` type security scheme in OpenAPI 2 and OpenAPI 3.

x-amazon-apigateway-authtype example

The following OpenAPI 3 example defines IAM authorization across multiple resources in a REST API or HTTP API:

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "openapi3",
    "version" : "1.0"
  },
  "paths" : {
    "/operation1" : {
      "get" : {
        "responses" : {
          "default" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "sigv4Reference" : [ ]
        } ]
      }
    },
    "/operation2" : {
      "get" : {
        "responses" : {
          "default" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "sigv4Reference" : [ ]
        } ]
      }
    }
  },
  "components" : {
    "securitySchemes" : {
      "sigv4Reference" : {
        "type" : "apiKey",
        "name" : "Authorization",
        "in" : "header",
        "x-amazon-apigateway-authtype": "awsSigv4"
      }
    }
  }
}
```

The following OpenAPI 3 example defines a Lambda authorizer with a custom scheme for a REST API:

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "openapi3 for REST API",
    "version" : "1.0"
  },
  "paths" : {
    "/protected-by-lambda-authorizer" : {
      "get" : {
        "responses" : {
          "200" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "myAuthorizer" : [ ]
        } ]
      }
    }
  },
  "components" : {
    "securitySchemes" : {
      "myAuthorizer" : {
        "type" : "apiKey",
        "name" : "Authorization",
        "in" : "header",
        "x-amazon-apigateway-authorizer" : {
          "identitySource" : "method.request.header.Authorization",
          "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:account-id:function:function-name/invocations",
          "authorizerResultTtlInSeconds" : 300,
          "type" : "request",
          "enableSimpleResponses" : false
        },
        "x-amazon-apigateway-authtype": "Custom scheme with corporate claims"
      }
    }
  },
  "x-amazon-apigateway-importexport-version" : "1.0"
}
```

See also

authorizer.authType

x-amazon-apigateway-binary-media-types property

Specifies the list of binary media types to be supported by API Gateway, such as `application/octet-stream` and `image/jpeg`. This extension is a JSON array. It should be included as a top-level vendor extension to the OpenAPI document.

x-amazon-apigateway-binary-media-types example

The following example shows the encoding lookup order of an API.

```
"x-amazon-apigateway-binary-media-types": [ "application/octet", "image/jpeg" ]
```

x-amazon-apigateway-documentation object

Defines the documentation parts to be imported into API Gateway. This object is a JSON object containing an array of the `DocumentationPart` instances.

Properties

Property name	Type	Description
`documentationParts`	`Array`	An array of the exported or imported `DocumentationPart` instances.
`version`	`String`	The version identifier of the snapshot of the exported documentation parts.

x-amazon-apigateway-documentation example

The following example of the API Gateway extension to OpenAPI defines `DocumentationParts` instances to be imported to or exported from an API in API Gateway.

```
{ ...
  "x-amazon-apigateway-documentation": {
    "version": "1.0.3",
    "documentationParts": [
       {
         "location": {
           "type": "API"
       },
        "properties": {
          "description": "API description",
          "info": {
            "description": "API info description 4",
            "version": "API info version 3"
          }
        }
      },
      {
         … // Another DocumentationPart instance
      }
    ]
  }
}
```

x-amazon-apigateway-endpoint-configuration object

Specifies details of the endpoint configuration for an API. This extension is an extended property of the OpenAPI Operation object. This object should be present in top-level vendor extensions for Swagger 2.0. For OpenAPI 3.0, it should be present under the vendor extensions of the Server object.

Properties

Property name	Type	Description
`disableExecuteApiEndpoint`	Boolean	Specifies whether clients can invoke your API by using the default `execute-api` endpoint. By default, clients can invoke your API with the default `https://{api_id}.execute-api.{region}.amazonaws.com` endpoint. To require that clients use a custom domain name to invoke your API, specify `true`.
`vpcEndpointIds`	An array of `String`	A list of VpcEndpoint identifiers against which to create Route 53 alias records for a REST API. It is only supported for REST APIs the `PRIVATE` endpoint type.

x-amazon-apigateway-endpoint-configuration examples

The following example associates specified VPC endpoints to the REST API.

```
"x-amazon-apigateway-endpoint-configuration": {
    "vpcEndpointIds": ["vpce-0212a4ababd5b8c3e", "vpce-01d622316a7df47f9"]
}
```

The following example disables the default endpoint for an API.

```
"x-amazon-apigateway-endpoint-configuration": {
    "disableExecuteApiEndpoint": true
}
```

x-amazon-apigateway-gateway-responses object

Defines the gateway responses for an API as a string-to-GatewayResponse map of key-value pairs. The extension applies to the root-level OpenAPI structure.

Properties

Property name	Type	Description
responseType	x-amazon-apigateway-gateway-responses.gatewayResponse	A `GatewayResponse` for the specified *responseType*.

x-amazon-apigateway-gateway-responses example

The following API Gateway extension to OpenAPI example defines a GatewayResponses map that contains two GatewayResponse instances—one for the `DEFAULT_4XX` type and another for the `INVALID_API_KEY` type.

```
{
  "x-amazon-apigateway-gateway-responses": {
    "DEFAULT_4XX": {
      "responseParameters": {
        "gatewayresponse.header.Access-Control-Allow-Origin": "'domain.com'"
      },
      "responseTemplates": {
        "application/json": "{\"message\": test 4xx b }"
      }
    },
    "INVALID_API_KEY": {
      "statusCode": "429",
      "responseTemplates": {
        "application/json": "{\"message\": test forbidden }"
      }
    }
  }
}
```

x-amazon-apigateway-gateway-responses.gatewayResponse object

Defines a gateway response of a given response type, including the status code, any applicable response parameters, or response templates.

Properties

Property name	Type	Description
responseParameters	x-amazon-apigateway-gateway-responses.responseParameters	Specifies the GatewayResponse parameters, namely the header parameters. The parameter values can take any incoming request parameter value or a static custom value.
responseTemplates	x-amazon-apigateway-gateway-responses.responseTemplates	Specifies the mapping templates of the gateway response. The templates are not processed by the VTL engine.
statusCode	string	An HTTP status code for the gateway response.

x-amazon-apigateway-gateway-responses.gatewayResponse example

The following example of the API Gateway extension to OpenAPI defines a GatewayResponse to customize the INVALID_API_KEY response to return the status code of 456, the incoming request's api-key header value, and a "Bad api-key" message.

```
    "INVALID_API_KEY": {
      "statusCode": "456",
      "responseParameters": {
        "gatewayresponse.header.api-key": "method.request.header.api-key"
      },
      "responseTemplates": {
        "application/json": "{\"message\": \"Bad api-key\" }"
      }
    }
```

x-amazon-apigateway-gateway-responses.responseParameters object

Defines a string-to-string map of key-value pairs to generate gateway response parameters from the incoming request parameters or using literal strings. Supported only for REST APIs.

Properties

Property name	Type	Description
gatewayresponse. *param-position* .*param-name*	string	*param-position* can be header, path, or querystring . For more information, see Map method request data to integration request parameters.

x-amazon-apigateway-gateway-responses.responseParameters example

The following OpenAPI extensions example shows a GatewayResponse response parameter mapping expression to enable CORS support for resources on the *.example.domain domains.

```
      "responseParameters": {
        "gatewayresponse.header.Access-Control-Allow-Origin": '*.example.domain',
        "gatewayresponse.header.from-request-header" : method.request.header.Accept,
        "gatewayresponse.header.from-request-path" : method.request.path.petId,
        "gatewayresponse.header.from-request-query" : method.request.querystring.qname
      }
```

x-amazon-apigateway-gateway-responses.responseTemplates object

Defines GatewayResponse mapping templates, as a string-to-string map of key-value pairs, for a given gateway response. For each key-value pair, the key is the content type. For example, "application/json" and the value is a stringified mapping template for simple variable substitutions. A `GatewayResponse` mapping template isn't processed by the Velocity Template Language (VTL) engine.

Properties

Property name	Type	Description
content-type	`string`	A `GatewayResponse` body mapping template supporting only simple variable substitution to customize a gateway response body.

x-amazon-apigateway-gateway-responses.responseTemplates example

The following OpenAPI extensions example shows a GatewayResponse mapping template to customize an API Gateway–generated error response into an app-specific format.

```

      "responseTemplates": {
        "application/json": "{ \"message\": $context.error.messageString, \"type\": 
$context.error.responseType, \"statusCode\": '488' }"
      }
```

The following OpenAPI extensions example shows a GatewayResponse mapping template to override an API Gateway–generated error response with a static error message.

```
      "responseTemplates": {
        "application/json": "{ \"message\": 'API-specific errors' }"
      }
```

x-amazon-apigateway-importexport-version

Specifies the version of the API Gateway import and export algorithm for HTTP APIs. Currently, the only supported value is `1.0`. To learn more, see exportVersion in the *API Gateway Version 2 API Reference*.

x-amazon-apigateway-importexport-version example

The following example sets the import and export version to `1.0`.

```
{
    "openapi": "3.0.1",
    "x-amazon-apigateway-importexport-version": "1.0",
    "info": { ...
```

x-amazon-apigateway-integration object

Specifies details of the backend integration used for this method. This extension is an extended property of the OpenAPI Operation object. The result is an API Gateway integration object.

Properties

Property name	Type	Description
`cacheKeyParameters`	An array of `string`	A list of request parameter s whose values are to be cached.
`cacheNamespace`	`string`	An API-specific tag group of related cached parameters.
`connectionId`	`string`	The ID of a VpcLink for the private integration.

Property name	Type	Description
connectionType	string	The integration connection type. The valid value is "VPC_LINK" for private integration or "INTERNET" , otherwise.
credentials	string	For AWS IAM role-based credentials, specify the ARN of an appropriate IAM role. If unspecified, credentials default to resource-based permissions that must be added manually to allow the API to access the resource. For more information, see Granting Permissions Using a Resource Policy. Note: When using IAM credentials, make sure that AWS STS Regional endpoints are enabled for the Region where this API is deployed for best performance.

Property name	Type	Description
contentHandling	string	Request payload encoding conversion types. Valid values are 1) CONVERT_TO_TEXT , for converting a binary payload into a base64-encoded string or converting a text payload into a utf-8-encoded string or passing through the text payload natively without modification, and 2) CONVERT_TO_BINARY , for converting a text payload into a base64-decoded blob or passing through a binary payload natively without modification.
httpMethod	string	The HTTP method used in the integration request. For Lambda function invocations, the value must be POST.
integrationSubtype	string	Specifies the integration subtype for an AWS service integration. Supported only for HTTP APIs. For supported integration subtypes, see the section called "AWS service integrations reference".

Property name	Type	Description
`passthroughBehavior`	`string`	Specifies how a request payload of unmapped content type is passed through the integration request without modification. Supported values are `when_no_templates` , `when_no_match` , and `never`. For more information, see Integration.passthroughBehavior.
`payloadFormatVersion`	`string`	Specifies the format of the payload sent to an integration. Required for HTTP APIs. For HTTP APIs, supported values for Lambda proxy integrations are `1.0` and `2.0`. For all other integrations, `1.0` is the only supported value. To learn more, see the section called "AWS Lambda integrations" and the section called "AWS service integrations reference".

Property name	Type	Description
`requestParameters`	x-amazon-apigateway-integration.requestParameters object	For REST APIs, specifies mappings from method request parameters to integration request parameters. Supported request parameters are `querystring` , `path`, `header`, and `body`. For HTTP APIs, request parameters are a key-value map specifying parameters that are passed to `AWS_PROXY` integrations with a specified `integrationSubtype` . You can provide static values, or map request data, stage variables, or context variables that are evaluated at runtime. To learn more, see the section called "AWS service integrations".
`requestTemplates`	x-amazon-apigateway-integration.requestTemplates object	Mapping templates for a request payload of specified MIME types.
`responses`	x-amazon-apigateway-integration.responses object	Defines the method's responses and specifies desired parameter mappings or payload mappings from integration responses to method responses.

Property name	Type	Description
timeoutInMillis	integer	Integration timeouts between 50 ms and 29,000 ms.
type	string	The type of integration with the specified backend. Valid values are: • http or http_proxy , for integration with an HTTP backend. • aws_proxy , for integration with AWS Lambda functions. • aws, for integration with AWS Lambda functions or other AWS services, such as Amazon DynamoDB, Amazon Simple Notification Service, or Amazon Simple Queue Service. • mock, for integration with API Gateway without invoking any backend. For more information about the integration types, see integration:type.
tlsConfig	the section called “x-amazon-apigateway-integration.tlsConfig”	Specifies the TLS configuration for an integration.

Property name	Type	Description
`uri`	`string`	The endpoint URI of the backend. For integrations of the `aws` type, this is an ARN value. For the HTTP integration, this is the URL of the HTTP endpoint including the `https` or `http` scheme.

x-amazon-apigateway-integration examples

For HTTP APIs, you can define integrations in the components section of your OpenAPI definition. To learn more, see x-amazon-apigateway-integrations object.

```
"x-amazon-apigateway-integration": {
    "$ref": "#/components/x-amazon-apigateway-integrations/integration1"
}
```

The following example creates an integration with a Lambda function. For demonstration purposes, the sample mapping templates shown in `requestTemplates` and `responseTemplates` of the examples below are assumed to apply to the following JSON-formatted payload: `{ "name":"value_1", "key":"value_2", "redirect": {"url" :"..."} }` to generate a JSON output of `{ "stage":"value_1", "user-id":"value_2" }` or an XML output of `<stage>value_1</stage>`.

```

"x-amazon-apigateway-integration" : {
    "type" : "aws",
    "uri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:012345678901:function:HelloWorld/invocations",
    "httpMethod" : "POST",
    "credentials" : "arn:aws:iam::012345678901:role/apigateway-invoke-lambda-exec-
role",
    "requestTemplates" : {
                "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
                "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
```

```
    },
    "requestParameters" : {
        "integration.request.path.stage" : "method.request.querystring.version",
        "integration.request.querystring.provider" :
 "method.request.querystring.vendor"
    },
    "cacheNamespace" : "cache namespace",
    "cacheKeyParameters" : [],
    "responses" : {
        "2\\d{2}" : {
            "statusCode" : "200",
            "responseParameters" : {
                "method.response.header.requestId" : "integration.response.header.cid"
            },
            "responseTemplates" : {
                "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
                "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
            }
        },
        "302" : {
            "statusCode" : "302",
            "responseParameters" : {
                "method.response.header.Location" :
 "integration.response.body.redirect.url"
            }
        },
        "default" : {
            "statusCode" : "400",
            "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static value'"
            }
        }
    }
}
```

Note that double quotes (") for the JSON string in the mapping templates must be string-escaped (\").

x-amazon-apigateway-integrations object

Defines a collection of integrations. You can define integrations in the components section of your OpenAPI definition, and reuse the integrations for multiple routes. Supported only for HTTP APIs.

Properties

Property name	Type	Description
integration	x-amazon-apigateway-integration object	A collection of integration objects.

x-amazon-apigateway-integrations example

The following example creates an HTTP API that defines two integrations, and references the integrations by using `$ref": "#/components/x-amazon-apigateway-integrations/`*`integration-name`*.

```
{
  "openapi": "3.0.1",
  "info":
    {
      "title": "Integrations",
      "description": "An API that reuses integrations",
      "version": "1.0"
    },
  "servers": [
  {
    "url": "https://example.com/{basePath}",
    "description": "The production API server",
    "variables":
      {
        "basePath":
          {
            "default": "example/path"
          }
      }
  }],
  "paths":
    {
      "/":
        {
          "get":
            {
              "x-amazon-apigateway-integration":
                {
                  "$ref": "#/components/x-amazon-apigateway-integrations/integration1"

                }
            }
        },
       "/pets":
        {
          "get":
            {
              "x-amazon-apigateway-integration":
                {
                  "$ref": "#/components/x-amazon-apigateway-integrations/integration1"

                }
            }
        },
       "/checkout":
        {
          "get":
            {
              "x-amazon-apigateway-integration":
                {
                  "$ref": "#/components/x-amazon-apigateway-integrations/integration2"
                }
            }
        }
    },
    "components": {
      "x-amazon-apigateway-integrations":
        {
          "integration1":
            {
              "type": "aws_proxy",
              "httpMethod": "POST",
              "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
              "passthroughBehavior": "when_no_templates",
              "payloadFormatVersion": "1.0"
            },
          "integration2":
            {
              "type": "aws_proxy",
              "httpMethod": "POST",
              "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:example-function/invocations",
              "passthroughBehavior": "when_no_templates",
              "payloadFormatVersion" : "1.0"
            }
        }
    }
}
```

x-amazon-apigateway-integration.requestTemplates object

Specifies mapping templates for a request payload of the specified MIME types.

Properties

Property name	Type	Description
MIME type	string	An example of the MIME type is application/json . For information about creating a mapping template, see PetStore mapping template.

x-amazon-apigateway-integration.requestTemplates example

The following example sets mapping templates for a request payload of the application/json and application/xml MIME types.

```
"requestTemplates" : {
    "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name\",
 \"user-id\": \"$root.key\" }",
    "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
}
```

x-amazon-apigateway-integration.requestParameters object

For REST APIs, specifies mappings from named method request parameters to integration request parameters. The method request parameters must be defined before being referenced.

For HTTP APIs, specifies parameters that are passed to `AWS_PROXY` integrations with a specified `integrationSubtype`.

Properties

Property name	Type	Description
`integration.request. *<param-type>* .*<param-name>*`	`string`	For REST APIs, the value is typically a predefined method request parameter of the `method.request. *<param-type>* .*<param-name>*` format, where `<param-type>` can be `querystring` , `path`, `header`, or `body`. However, `$context. *VARIABLE_NAME*` , `$stageVariables. *VARIABLE_NAME*` , and *`STATIC_VALUE`* are also valid. For the body parameter , the `<param-name>` is a JSON path expression without the `$.` prefix.
`parameter`	`string`	For HTTP APIs, request parameters are a key-value map specifying

Property name	Type	Description
		parameters that are passed to `AWS_PROXY` integrations with a specified `integrationSubtype` . You can provide static values, or map request data, stage variables , or context variables that are evaluated at runtime. To learn more, see the section called "AWS service integrations".

`x-amazon-apigateway-integration.requestParameters` example

The following request parameter mappings example translates a method request's query (`version`), header (`x-user-id`), and path (`service`) parameters to the integration request's query (`stage`), header (`x-userid`), and path parameters (op), respectively.

Note

If you're creating resources through OpenAPI or AWS CloudFormation, static values should be enclosed in single quotes.
To add this value from the console, enter `application/json` in the box, without quotation marks.

```
"requestParameters" : {
    "integration.request.querystring.stage" : "method.request.querystring.version",
    "integration.request.header.x-userid" : "method.request.header.x-user-id",
    "integration.request.path.op" : "method.request.path.service"
},
```

x-amazon-apigateway-integration.responses object

Defines the method's responses and specifies parameter mappings or payload mappings from integration responses to method responses.

Properties

Property name	Type	Description
Response status pattern	x-amazon-apigateway-integration.response object	Either a regular expression used to match the integration response to the method response, or `default` to catch any response that you haven't configured. For HTTP integrations, the regex applies to the integration response status code. For Lambda invocations, the regex applies to the `errorMessage` field of the error information object returned by AWS Lambda as a failure response body when the Lambda function execution throws an exception. **Note** The *Response status pattern* property name refers to a response status code or regular expression describing a group of response status codes. It does not correspond to

Property name	Type	Description
		any identifier of an IntegrationResponse resource in the API Gateway REST API.

x-amazon-apigateway-integration.responses example

The following example shows a list of responses from 2xx and 302 responses. For the 2xx response, the method response is mapped from the integration response's payload of the application/json or application/xml MIME type. This response uses the supplied mapping templates. For the 302 response, the method response returns a Location header whose value is derived from the redirect.url property on the integration response's payload.

```
"responses" : {
    "2\\d{2}" : {
        "statusCode" : "200",
        "responseTemplates" : {
            "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
            "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
        }
    },
    "302" : {
        "statusCode" : "302",
        "responseParameters" : {
            "method.response.header.Location": "integration.response.body.redirect.url"
        }
    }
}
```

x-amazon-apigateway-integration.response object

Defines a response and specifies parameter mappings or payload mappings from the integration response to the method response.

Properties

Property name	Type	Description
`statusCode`	`string`	HTTP status code for the method response; for example, `"200"`. This must correspond to a matching response in the OpenAPI Operation `responses` field.
`responseTemplates`	x-amazon-apigateway-integration.responseTemplates object	Specifies MIME type-specific mapping templates for the response's payload.
`responseParameters`	x-amazon-apigateway-integration.responseParameters object	Specifies parameter mappings for the response. Only the `header` and body parameters of the integration response can be mapped to the `header` parameters of the method.
`contentHandling`	`string`	Response payload encoding conversion types. Valid values are 1) `CONVERT_TO_TEXT` , for converting a binary payload into a base64-encoded string or converting a text payload into a `utf-8`-encoded string or passing through the text payload natively without modificat

Property name	Type	Description
		ion, and 2) CONVERT_TO_BINARY , for converting a text payload into a base64-decoded blob or passing through a binary payload natively without modification.

x-amazon-apigateway-integration.response example

The following example defines a 302 response for the method that derives a payload of the application/json or application/xml MIME type from the backend. The response uses the supplied mapping templates and returns the redirect URL from the integration response in the method's Location header.

```
{
    "statusCode" : "302",
    "responseTemplates" : {
         "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name
\", \"user-id\": \"$root.key\" }",
         "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
    },
    "responseParameters" : {
        "method.response.header.Location": "integration.response.body.redirect.url"
    }
}
```

x-amazon-apigateway-integration.responseTemplates object

Specifies mapping templates for a response payload of the specified MIME types.

Properties

Property name	Type	Description
MIME type	string	Specifies a mapping template to transform the integration response body to the method response body for a given MIME type. For information about creating a mapping template, see [PetStore mapping template](). An example of the *MIME type* is application/json.

x-amazon-apigateway-integration.responseTemplate example

The following example sets mapping templates for a request payload of the application/json and application/xml MIME types.

```
"responseTemplates" : {
    "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name\",
 \"user-id\": \"$root.key\" }",
    "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
}
```

x-amazon-apigateway-integration.responseParameters object

Specifies mappings from integration method response parameters to method response parameters. You can map header, body, or static values to the header type of the method response. Supported only for REST APIs.

Properties

Property name	Type	Description
method.response.header. *<param-name>*	string	The named parameter value can be derived from the header and body types of the integration response parameters.

x-amazon-apigateway-integration.responseParameters example

The following example maps body and header parameters of the integration response to two header parameters of the method response.

```
"responseParameters" : {
    "method.response.header.Location" : "integration.response.body.redirect.url",
    "method.response.header.x-user-id" : "integration.response.header.x-userid"
}
```

x-amazon-apigateway-integration.tlsConfig object

Specifies the TLS configuration for an integration.

Properties

Property name	Type	Description
insecureSkipVerification	Boolean	Supported only for REST APIs. Specifies whether or not API Gateway skips verification that the certificate for an integration endpoint is issued by a supported certificate authority. This isn't

Property name	Type	Description
		recommended, but it enables you to use certificates that are signed by private certificate authorities, or certificates that are self-signed. If enabled, API Gateway still performs basic certificate validation, which includes checking the certificate's expiration date, hostname, and presence of a root certificate authority. The root certificate belonging to the private authority must satisfy the following constraints: - x509 extension `keyUsage` must have `keyCertSign` . - x509 extension `basicConstraints` must have `CA:TRUE`. Supported only for HTTP and `HTTP_PROXY` integrations. ⚠ **Warning** Enabling `insecureSkipVerification` isn't recommended, especially for integrations with public HTTPS endpoints. If you enable `insecureSkipVerifi`

Property name	Type	Description
		`cation`, you increase the risk of man-in-the-middle attacks.
`serverNameToVerify`	`string`	Supported only for HTTP API private integrations. If you specify a server name, API Gateway uses it to verify the hostname on the integration's certificate. The server name is also included in the TLS handshake to support Server Name Indication (SNI) or virtual hosting.

x-amazon-apigateway-integration.tlsConfig examples

The following OpenAPI 3.0 example enables `insecureSkipVerification` for a REST API HTTP proxy integration.

```
"x-amazon-apigateway-integration": {
  "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
  "responses": {
     default": {
       "statusCode": "200"
      }
  },
  "passthroughBehavior": "when_no_match",
  "httpMethod": "ANY",
  "tlsConfig" : {
    "insecureSkipVerification" : true
  }
  "type": "http_proxy",
}
```

The following OpenAPI 3.0 example specifies a `serverNameToVerify` for an HTTP API private integration.

```
"x-amazon-apigateway-integration" : {
  "payloadFormatVersion" : "1.0",
  "connectionId" : "abc123",
  "type" : "http_proxy",
  "httpMethod" : "ANY",
  "uri" : "arn:aws:elasticloadbalancing:us-west-2:123456789012:listener/app/my-load-
balancer/50dc6c495c0c9188/0467ef3c8400ae65",
  "connectionType" : "VPC_LINK",
  "tlsConfig" : {
     "serverNameToVerify" : "example.com"
  }
}
```

x-amazon-apigateway-minimum-compression-size

Specifies the minimum compression size for a REST API. To enable compression, specify an integer between 0 and 10485760. To learn more, see Enabling payload compression for an API.

x-amazon-apigateway-minimum-compression-size example

The following example specifies a minimum compression size of `5242880` bytes for a REST API.

```
"x-amazon-apigateway-minimum-compression-size": 5242880
```

x-amazon-apigateway-policy

Specifies a resource policy for a REST API. To learn more about resource policies, see Controlling access to an API with API Gateway resource policies. For resource policy examples, see API Gateway resource policy examples.

`x-amazon-apigateway-policy` example

The following example specifies a resource policy for a REST API. The resource policy denies (blocks) incoming traffic to an API from a specified source IP address block. On import, `"execute-api:/*"` is converted to `arn:aws:execute-api:`*`region`*`:`*`account-id`*`:`*`api-id`*`/*`, using the current Region, your AWS account ID, and the current REST API ID.

```
"x-amazon-apigateway-policy": {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:SourceIp": "192.0.2.0/24"
                }
            }
        }
    ]
}
```

x-amazon-apigateway-request-validator property

Specifies a request validator, by referencing a *request_validator_name* of the x-amazon-apigateway-request-validators object map, to enable request validation on the containing API or a method. The value of this extension is a JSON string.

This extension can be specified at the API level or at the method level. The API-level validator applies to all of the methods unless it is overridden by the method-level validator.

x-amazon-apigateway-request-validator example

The following example applies the basic request validator at the API level while applying the parameter-only request validator on the POST /validation request.

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "x-amazon-apigateway-request-validators" : {
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  "x-amazon-apigateway-request-validator" : "basic",
  "paths": {
    "/validation": {
      "post": {
        "x-amazon-apigateway-request-validator" : "params-only",
       ...
     }
}
```

x-amazon-apigateway-request-validators object

Defines the supported request validators for the containing API as a map between a validator name and the associated request validation rules. This extension applies to a REST API.

Properties

Property name	Type	Description

request_validator_name	x-amazon-apigateway-request-validators.requestValidator object	Specifies the validation rules consisting of the named validator. For example:
		`"basic" : {`
		`"validateRequestBody" : true,`

Property name	Type	Description

		`"validateRequestParameters" : true`
		`},`
		To apply this validator to a specific method, reference the validator name (`basic`) as the value of the [x-amazon-apigateway-request-validator property](#) property.

`x-amazon-apigateway-request-validators` example

The following example shows a set of request validators for an API as a map between a validator name and the associated request validation rules.

OpenAPI 2.0

```
{
  "swagger": "2.0",
  ...
  "x-amazon-apigateway-request-validators" : {
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  ...
}
```

x-amazon-apigateway-request-validators.requestValidator object

Specifies the validation rules of a request validator as part of the [x-amazon-apigateway-request-validators object](#) map definition.

Properties

Property name	Type	Description
`validateRequestBody`	`Boolean`	Specifies whether to validate the request body (`true`) or not (`false`).
`validateRequestParameters`	`Boolean`	Specifies whether to validate the required request parameters (`true`) or not (`false`).

`x-amazon-apigateway-request-validators.requestValidator` example

The following example shows a parameter-only request validator:

```
"params-only": {
    "validateRequestBody" : false,
    "validateRequestParameters" : true
}
```

x-amazon-apigateway-tag-value property

Specifies the value of an [AWS tag](#) for an HTTP API. You can use the `x-amazon-apigateway-tag-value` property as part of the root-level [OpenAPI tag object](#) to specify AWS tags for an HTTP API. If you specify a tag name without the `x-amazon-apigateway-tag-value` property, API Gateway creates a tag with an empty string for a value.

To learn more about tagging, see [Tagging your API Gateway resources](#).

Properties

Property name	Type	Description
`name`	`String`	Specifies the tag key.
`x-amazon-apigateway-tag-value`	`String`	Specifies the tag value.

`x-amazon-apigateway-tag-value` example

The following example specifies two tags for an HTTP API:

- "Owner": "Admin"
- "Prod": ""

```
"tags": [
    {
      "name": "Owner",
      "x-amazon-apigateway-tag-value": "Admin"
    },
    {
      "name": "Prod"
    }
]
```

Security in Amazon API Gateway

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 Amazon API Gateway, 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 API Gateway. The following topics show you how to configure API Gateway to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your API Gateway resources.

For more information, see Security Overview of Amazon API Gateway.

Topics

Data protection in Amazon API Gateway

The AWS shared responsibility model applies to data protection in Amazon API Gateway. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with API Gateway or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption in Amazon API Gateway

Data protection refers to protecting data while in transit (as it travels to and from API Gateway) and at rest (while it is stored in AWS).

Data encryption at rest in Amazon API Gateway

If you choose to enable caching for a REST API, you can enable cache encryption. To learn more, see Enabling API caching to enhance responsiveness.

For more information about data protection, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

Data encryption in transit in Amazon API Gateway

The APIs created with Amazon API Gateway expose HTTPS endpoints only. API Gateway doesn't support unencrypted (HTTP) endpoints.

API Gateway manages the certificates for default `execute-api` endpoints. If you configure a custom domain name, you specify the certificate for the domain name. As a best practice, don't pin certificates.

For greater security, you can choose a minimum Transport Layer Security (TLS) protocol version to be enforced for your API Gateway custom domain. WebSocket APIs and HTTP APIs support only TLS 1.2. To learn more, see Choosing a security policy for your custom domain in API Gateway.

You can also set up a Amazon CloudFront distribution with a custom SSL certificate in your account and use it with Regional APIs. You can then configure the security policy for the CloudFront distribution with TLS 1.1 or higher based on your security and compliance requirements.

For more information about data protection, see Protecting your REST API and the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

Internetwork traffic privacy

Using Amazon API Gateway, you can create private REST APIs that can be accessed only from your Amazon Virtual Private Cloud (VPC). The VPC uses an interface VPC endpoint, which is an endpoint network interface that you create in your VPC. Using resource policies, you can allow or deny access to your API from selected VPCs and VPC endpoints, including across AWS accounts. Each endpoint can be used to access multiple private APIs. You can also use AWS Direct Connect to establish a connection from an on-premises network to Amazon VPC and access your private API over that connection. In all cases, traffic to your private API uses secure connections and does not leave the Amazon network; it is isolated from the public internet. To learn more, see the section called “Private APIs”.

Identity and access management for Amazon API Gateway

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 API Gateway resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in API Gateway.

Service user – If you use the API Gateway service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more API Gateway 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 API Gateway, see Troubleshooting Amazon API Gateway identity and access.

Service administrator – If you're in charge of API Gateway resources at your company, you probably have full access to API Gateway. It's your job to determine which API Gateway features and resources your service users 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 API Gateway, see How Amazon API Gateway works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to API Gateway. To view example API Gateway identity-based policies that you can use in IAM, see Amazon API Gateway identity-based policy examples.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signing AWS API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might 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 Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one 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 don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

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:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using

an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) 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*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

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, group of users, or role. These policies control what actions users and roles 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. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, 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 an 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 Amazon API Gateway works with IAM

Before you use IAM to manage access to API Gateway, you should understand what IAM features are available to use with API Gateway. To get a high-level view of how API Gateway and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

API Gateway identity-based policies

With IAM identity-based policies, you can specify which actions and resources are allowed or denied as well as the conditions under which actions are allowed or denied. API Gateway supports specific actions, resources, and condition keys. For more information about the API Gateway-specific actions, resources, and condition keys, see Actions, resources, and condition keys for Amazon API Gateway Management and Actions, resources, and condition keys for Amazon API

Gateway Management V2. For information about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

The following example shows an identity-based policy that allows a user to create or update only private REST APIs. For more examples, see the section called “Identity-based policy examples”.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ScopeToPrivateApis",
      "Effect": "Allow",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/restapis",
        "arn:aws:apigateway:us-east-1::/restapis/??????????"
      ],
      "Condition": {
        "ForAllValues:StringEqualsIfExists": {
          "apigateway:Request/EndpointType": "PRIVATE",
          "apigateway:Resource/EndpointType": "PRIVATE"
        }
      }
    },
    {
      "Sid": "AllowResourcePolicyUpdates",
      "Effect": "Allow",
      "Action": [
          "apigateway:UpdateRestApiPolicy"
      ],
      "Resource": [
          "arn:aws:apigateway:us-east-1::/restapis/*"
      ]
    }
  ]
}
```

Actions

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy.

Policy actions in API Gateway use the following prefix before the action: `apigateway:`. Policy statements must include either an `Action` or `NotAction` element. API Gateway defines its own set of actions that describe tasks that you can perform with this service.

The API-managing `Action` expression has the format `apigateway:`*`action`*, where *`action`* is one of the following API Gateway actions: **GET**, **POST**, **PUT**, **DELETE**, **PATCH** (to update resources), or *****, which is all of the previous actions.

Some examples of the `Action` expression include:

- **`apigateway:*`** for all API Gateway actions.
- **`apigateway:GET`** for just the GET action in API Gateway.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "apigateway:action1",
      "apigateway:action2"
```

For information about HTTP verbs to use for specific API Gateway operations, see Amazon API Gateway Version 1 API Reference (REST APIs) and Amazon API Gateway Version 2 API Reference (WebSocket and HTTP APIs).

For more information, see the section called “Identity-based policy examples”.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

API Gateway resources have the following ARN format:

```
arn:aws:apigateway:region::resource-path-specifier
```

For example, to specify a REST API with the id *`api-id`* and its sub-resources, such as authorizers in your statement, use the following ARN:

```
"Resource": "arn:aws:apigateway:us-east-2::/restapis/api-id/*"
```

To specify all REST APIs and sub-resources that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:apigateway:us-east-2::/restapis/*"
```

For a list of API Gateway resource types and their ARNs, see API Gateway Amazon Resource Name (ARN) reference.

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

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 supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

API Gateway defines its own set of condition keys and also supports using some global condition keys. For a list of API Gateway condition keys, see Condition Keys for Amazon API Gateway in the *IAM User Guide*. For information about which actions and resources you can use with a condition key, see Actions Defined by Amazon API Gateway.

For information about tagging, including attribute-based access control, see *Tagging*.

Examples

For examples of API Gateway identity-based policies, see Amazon API Gateway identity-based policy examples.

API Gateway resource-based policies

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on the API Gateway resource and under what conditions. API Gateway supports resource-based permissions policies for REST APIs. You use resource policies to control who can invoke a REST API. For more information, see the section called "Use API Gateway resource policies".

Examples

For examples of API Gateway resource-based policies, see API Gateway resource policy examples.

Authorization based on API Gateway tags

You can attach tags to API Gateway resources or pass tags in a request to API Gateway. To control access based on tags, you provide tag information in the condition element of a policy using the `apigateway:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys. For more information about tagging API Gateway resources, see the section called "Attribute-based access control".

For an examples of identity-based policies for limiting access to a resource based on the tags on that resource, see Using tags to control access to API Gateway REST API resources.

API Gateway IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with API Gateway

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.

API Gateway supports using temporary credentials.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

API Gateway supports service-linked roles. For information about creating or managing API Gateway service-linked roles, see Using service-linked roles for API Gateway.

Service roles

A service can assume a service role on your behalf. A service 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, so an administrator can change the permissions for this role. However, doing so might break the functionality of the service.

API Gateway supports service roles.

Amazon API Gateway identity-based policy examples

By default, IAM users and roles don't have permission to create or modify API Gateway resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS SDKs. 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.

For information about how to create IAM policies, see Creating Policies on the JSON Tab in the *IAM User Guide*. For information about the actions, resources, and conditions specific to API Gateway, see Actions, resources, and condition keys for Amazon API Gateway Management and Actions, resources, and condition keys for Amazon API Gateway Management V2.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete API Gateway resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and

functional policies. For more information, see IAM Access Analyzer policy validation in the *IAM User Guide*.

- **Require multi-factor authentication (MFA) –** If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Simple read permissions

This example policy gives a user permission to get information about all of the resources of an HTTP or WebSocket API with the identifier of a123456789 in the AWS Region of us-east-1. The resource arn:aws:apigateway:*us-east-1*::/apis/a123456789/* includes all sub-resources of the API such as authorizers and deployments.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis/a123456789/*"
      ]
    }
  ]
}
```

Create only REQUEST or JWT authorizers

This example policy allows a user to create APIs with only REQUEST or JWT authorizers, including through import. In the Resource section of the policy, arn:aws:apigateway:us-east-1::/apis/?????????? requires that resources have a maximum of 10 characters, which excludes sub-resources of an API. This example uses ForAllValues in the Condition section because users can create multiple authorizers at once by importing an API.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "OnlyAllowSomeAuthorizerTypes",
      "Effect": "Allow",
      "Action": [
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:PATCH"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/??????????",
        "arn:aws:apigateway:us-east-1::/apis/*/authorizers",
        "arn:aws:apigateway:us-east-1::/apis/*/authorizers/*"
      ],
      "Condition": {
        "ForAllValues:StringEqualsIfExists": {
          "apigateway:Request/AuthorizerType": [
            "REQUEST",
            "JWT"
          ]
        }
      }
    }
  ]
}
```

Require that the default `execute-api` endpoint is disabled

This example policy allows users to create, update or import an API, with the requirement that `DisableExecuteApiEndpoint` is `true`. When `DisableExecuteApiEndpoint` is `true`, clients can't use the default `execute-api` endpoint to invoke an API.

We use the `BoolIfExists` condition to handle a call to update an API that doesn't have the `DisableExecuteApiEndpoint` condition key populated. When a user attempts to create or import an API, the `DisableExecuteApiEndpoint` condition key is always populated.

Because the `apis/*` resource also captures sub resources such as authorizers or methods, we explicitly scope it to just APIs with a `Deny` statement.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DisableExecuteApiEndpoint",
      "Effect": "Allow",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/*"
      ],
      "Condition": {
        "BoolIfExists": {
          "apigateway:Request/DisableExecuteApiEndpoint": true
        }
      }
    },
    {
      "Sid": "ScopeDownToJustApis",
      "Effect": "Deny",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis/*/*"
      ]
    }
  ]
}
```

Allow users to create or update only private REST APIs

This example policy uses condition keys to require that a user creates only `PRIVATE` APIs, and to prevent updates that might change an API from `PRIVATE` to another type, such as `REGIONAL`.

We use `ForAllValues` to require that every `EndpointType` added to an API is `PRIVATE`. We use a resource condition key to allow any update to an API as long as it's `PRIVATE`. `ForAllValues` applies only if a condition key is present.

We use the non-greedy matcher (?) to explicitly match against API IDs to prevent allowing non-API resources such as authorizers.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ScopePutToPrivateApis",
            "Effect": "Allow",
            "Action": [
                "apigateway:PUT"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis",
                "arn:aws:apigateway:us-east-1::/restapis/??????????"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "apigateway:Resource/EndpointType": "PRIVATE"
                }
            }
        },
        {
            "Sid": "ScopeToPrivateApis",
            "Effect": "Allow",
            "Action": [
                "apigateway:DELETE",
                "apigateway:PATCH",
                "apigateway:POST"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis",
                "arn:aws:apigateway:us-east-1::/restapis/??????????"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "apigateway:Request/EndpointType": "PRIVATE",
                    "apigateway:Resource/EndpointType": "PRIVATE"
                }
            }
        },
        {
            "Sid": "AllowResourcePolicyUpdates",
            "Effect": "Allow",
            "Action": [
                "apigateway:UpdateRestApiPolicy"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis/*"
            ]
        }
    ]
}
```

Require that API routes have authorization

This policy causes attempts to create or update a route (including through import) to fail if the route has no authorization. `ForAnyValue` evaluates to false if the key is not present, such as when a route is not being created or updated. We use `ForAnyValue` because multiple routes can be created through import.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdatesOnApisAndRoutes",
      "Effect": "Allow",
      "Action": [
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/??????????",
        "arn:aws:apigateway:us-east-1::/apis/*/routes",
        "arn:aws:apigateway:us-east-1::/apis/*/routes/*"
      ]
    },
    {
      "Sid": "DenyUnauthorizedRoutes",
      "Effect": "Deny",
      "Action": [
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/*"
      ],
      "Condition": {
        "ForAnyValue:StringEqualsIgnoreCase": {
          "apigateway:Request/RouteAuthorizationType": "NONE"
        }
      }
    }
  ]
}
```

Prevent a user from creating or updating a VPC link

This policy prevents a user from creating or updating a VPC link. A VPC link enables you to expose resources within an Amazon VPC to clients outside of the VPC.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyVPCLink",
      "Effect": "Deny",
      "Action": [
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:PATCH"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/vpclinks",
        "arn:aws:apigateway:us-east-1::/vpclinks/*"
      ]
    }
  ]
}
```

Amazon API Gateway resource-based policy examples

For resource-based policy examples, see the section called “API Gateway resource policy examples”.

Troubleshooting Amazon API Gateway identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with API Gateway and IAM.

Topics

I am not authorized to perform an action in API Gateway

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *my-example-widget* resource but doesn't have the fictional `apigateway:`*GetWidget* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 apigateway:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *my-example-widget* resource by using the `apigateway:`*GetWidget* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to API Gateway.

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 API Gateway. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my API Gateway 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 API Gateway supports these features, see How Amazon API Gateway works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Using service-linked roles for API Gateway

Amazon API Gateway uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to API Gateway. Service-linked roles are predefined by API Gateway and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up API Gateway easier because you don't have to manually add the necessary permissions. API Gateway defines the permissions of its service-linked roles, and unless defined otherwise, only API Gateway can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting the related resources. This protects your API Gateway resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for API Gateway

API Gateway uses the service-linked role named **AWSServiceRoleForAPIGateway** – Allows API Gateway to access Elastic Load Balancing, Amazon Data Firehose, and other service resources on your behalf.

The AWSServiceRoleForAPIGateway service-linked role trusts the following services to assume the role:

- `ops.apigateway.amazonaws.com`

The role permissions policy allows API Gateway to complete the following actions on the specified resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticloadbalancing:AddListenerCertificates",
                "elasticloadbalancing:RemoveListenerCertificates",
                "elasticloadbalancing:ModifyListener",
                "elasticloadbalancing:DescribeListeners",
                "elasticloadbalancing:DescribeLoadBalancers",
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "logs:CreateLogDelivery",
                "logs:GetLogDelivery",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:ListLogDeliveries",
                "servicediscovery:DiscoverInstances"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "firehose:DescribeDeliveryStream",
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Resource": "arn:aws:firehose:*:*:deliverystream/amazon-apigateway-*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "acm:DescribeCertificate",
                "acm:GetCertificate"
            ],
            "Resource": "arn:aws:acm:*:*:certificate/*"
        },
        {
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterfacePermission",
            "Resource": "arn:aws:ec2:*:*:network-interface/*"
        },
        {
            "Effect": "Allow",
            "Action": "ec2:CreateTags",
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "aws:TagKeys": [
                        "Owner",
                        "VpcLinkId"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:ModifyNetworkInterfaceAttribute",
                "ec2:DeleteNetworkInterface",
                "ec2:AssignPrivateIpAddresses",
                "ec2:CreateNetworkInterface",
                "ec2:DeleteNetworkInterfacePermission",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeNetworkInterfaceAttribute",
                "ec2:DescribeVpcs",
                "ec2:DescribeNetworkInterfacePermissions",
                "ec2:UnassignPrivateIpAddresses",
                "ec2:DescribeSubnets",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "servicediscovery:GetNamespace",
            "Resource": "arn:aws:servicediscovery:*:*:namespace/*"
        },
        {
            "Effect": "Allow",
            "Action": "servicediscovery:GetService",
            "Resource": "arn:aws:servicediscovery:*:*:service/*"
        }
    ]
}
```

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating a service-linked role for API Gateway

You don't need to manually create a service-linked role. When you create an API, custom domain name, or VPC link in the AWS Management Console, the AWS CLI, or the AWS API, API Gateway creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create an API, custom domain name, or VPC link, API Gateway creates the service-linked role for you again.

Editing a service-linked role for API Gateway

API Gateway does not allow you to edit the AWSServiceRoleForAPIGateway service-linked role. After you create a service-linked role, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting a service-linked role for API Gateway

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

> **Note**
>
> If the API Gateway service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To delete API Gateway resources used by the AWSServiceRoleForAPIGateway

1. Open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Navigate to the API, custom domain name, or VPC link that uses the service-linked role.
3. Use the console to delete the resource.
4. Repeat the procedure to delete all APIs, custom domain names, or VPC links that use the service-linked role.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForAPIGateway service-linked role. For more information, see [Deleting a Service-Linked Role]() in the *IAM User Guide*.

Supported Regions for API Gateway service-linked roles

API Gateway supports using service-linked roles in all of the Regions where the service is available. For more information, see [AWS Service Endpoints]().

API Gateway updates to AWS managed policies

View details about updates to AWS managed policies for API Gateway since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the API Gateway [Document history]() page.

Change	Description	Date
Added `acm:GetCertificate` support to the `AWSServiceRoleForAPIGateway` policy.	The `AWSServiceRoleForAPIGateway` policy now includes permission to call the ACM `GetCertificate` API action.	July 12, 2021
API Gateway started tracking changes	API Gateway started tracking changes for its AWS managed policies.	July 12, 2021

Logging and monitoring in Amazon API Gateway

Monitoring is an important part of maintaining the reliability, availability, and performance of API Gateway and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your API Gateway resources and responding to potential incidents:

Amazon CloudWatch Logs

To help debug issues related to request execution or client access to your API, you can enable CloudWatch Logs to log API calls. For more information, see [the section called "CloudWatch logs"]().

Amazon CloudWatch Alarms

Using CloudWatch alarms, you watch a single metric over a time period that you specify. If the metric exceeds a given threshold, a notification is sent to an Amazon Simple Notification Service topic or AWS Auto Scaling policy. CloudWatch alarms do not invoke actions when a metric is in a particular state. Rather the state must have changed and been maintained for a specified number of periods. For more information, see the section called "CloudWatch metrics".

Access Logging to Firehose

To help debug issues related to client access to your API, you can enable Firehose to log API calls. For more information, see the section called "Firehose".

AWS CloudTrail

CloudTrail provides a record of actions taken by a user, role, or an AWS service in API Gateway. Using the information collected by CloudTrail, you can determine the request that was made to API Gateway, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, see the section called "Working with CloudTrail".

AWS X-Ray

X-Ray is an AWS service that gathers data about the requests that your application serves, and uses it to construct a service map that you can use to identify issues with your application and opportunities for optimization. For more information, see the section called "Setting up AWS X-Ray".

AWS Config

AWS Config provides a detailed view of the configuration of AWS resources in your account. You can see how resources are related, get a history of configuration changes, and see how relationships and configurations change over time. You can use AWS Config to define rules that evaluate resource configurations for data compliance. AWS Config rules represent the ideal configuration settings for your API Gateway resources. If a resource violates a rule and is flagged as noncompliant, AWS Config can alert you using an Amazon Simple Notification Service (Amazon SNS) topic. For details, see the section called "Working with AWS Config".

Logging Amazon API Gateway API calls using AWS CloudTrail

Amazon API Gateway is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all REST API calls for API Gateway service as events. The calls captured include calls from the API Gateway console and code calls to the API Gateway service APIs. Using the information collected by CloudTrail, you can determine the request that was made to API Gateway, the IP address from which the request was made, when it was made, and additional details.

> **Note**
>
> TestInvokeAuthorizer and TestInvokeMethod are not logged in CloudTrail.

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 user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

API Gateway management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

Amazon API Gateway logs all API Gateway actions as management events, except for TestInvokeAuthorizer and TestInvokeMethod. For a list of the Amazon API Gateway actions that API Gateway logs to CloudTrail, see the Amazon API Gateway API Reference.

API Gateway event example

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail event that demonstrates the API Gateway `GetResource` action:

```
{
    Records: [
        {
            eventVersion: "1.03",
            userIdentity: {
                type: "Root",
                principalId: "AKIAI44QH8DHBEXAMPLE",
                arn: "arn:aws:iam::123456789012:root",
                accountId: "123456789012",
                accessKeyId: "AKIAIOSFODNN7EXAMPLE",
                sessionContext: {
                    attributes: {
                        mfaAuthenticated: "false",
                        creationDate: "2015-06-16T23:37:58Z"
                    }
                }
            },
            eventTime: "2015-06-17T00:47:28Z",
            eventSource: "apigateway.amazonaws.com",
            eventName: "GetResource",
            awsRegion: "us-east-1",
            sourceIPAddress: "203.0.113.11",
            userAgent: "example-user-agent-string",
            requestParameters: {
                restApiId: "3rbEXAMPLE",
                resourceId: "5tfEXAMPLE",
                template: false
            },
            responseElements: null,
            requestID: "6d9c4bfc-148a-11e5-81b6-7577cEXAMPLE",
            eventID: "4d293154-a15b-4c33-9e0a-ff5eeEXAMPLE",
            readOnly: true,
            eventType: "AwsApiCall",
            recipientAccountId: "123456789012"
        },
        ... additional entries ...
    ]
}
```

For information about CloudTrail record contents, see CloudTrail record contents in the *AWS CloudTrail User Guide*.

Monitoring API Gateway API configuration with AWS Config

You can use AWS Config to record configuration changes made to your API Gateway API resources and send notifications based on resource changes. Maintaining a configuration change history for API Gateway resources is useful for operational troubleshooting, audit, and compliance use cases.

AWS Config can track changes to:

- **API stage configuration**, such as:
  - cache cluster settings
  - throttle settings
  - access log settings
  - the active deployment set on the stage
- **API configuration**, such as:
  - endpoint configuration
  - version
  - protocol
  - tags

In addition, the AWS Config Rules feature enables you to define configuration rules and automatically detect, track, and alert violations to these rules. By tracking changes to these resource configuration properties, you can also author change-triggered AWS Config rules for your API Gateway resources, and test your resource configurations against best practices.

You can enable AWS Config in your account by using the AWS Config console or the AWS CLI. Select the resource types for which you want to track changes. If you previously configured AWS Config to record all resource types, then these API Gateway resources will be automatically recorded in your account. Support for Amazon API Gateway in AWS Config is available in all AWS public regions and AWS GovCloud (US). For the full list of supported Regions, see Amazon API Gateway Endpoints and Quotas in the AWS General Reference.

Topics

- Supported resource types
- Setting up AWS Config
- Configuring AWS Config to record API Gateway resources
- Viewing API Gateway configuration details in the AWS Config console
- Evaluating API Gateway resources using AWS Config rules

Supported resource types

The following API Gateway resource types are integrated with AWS Config and are documented in AWS Config Supported AWS Resource Types and Resource Relationships:

- `AWS::ApiGatewayV2::Api` (WebSocket and HTTP API)
- `AWS::ApiGateway::RestApi` (REST API)
- `AWS::ApiGatewayV2::Stage` (WebSocket and HTTP API stage)
- `AWS::ApiGateway::Stage` (REST API stage)

For more information about AWS Config, see the AWS Config Developer Guide. For pricing information, see the AWS Config pricing information page.

> ⚠ **Important**
>
> If you change any of the following API properties after the API is deployed, you *must* redeploy the API to propagate the changes. Otherwise, you'll see the attribute changes in the AWS Config console, but the previous property settings will still be in effect; the API's runtime behavior will be unchanged.
>
> - **`AWS::ApiGateway::RestApi`** – `binaryMediaTypes`, `minimumCompressionSize`, `apiKeySource`
> - **`AWS::ApiGatewayV2::Api`** – `apiKeySelectionExpression`

Setting up AWS Config

To initially set up AWS Config, see the following topics in the AWS Config Developer Guide.

- Setting Up AWS Config with the Console

- Setting Up AWS Config with the AWS CLI

Configuring AWS Config to record API Gateway resources

By default, AWS Config records configuration changes for all supported types of regional resources that it discovers in the region in which your environment is running. You can customize AWS Config to record changes only for specific resource types, or changes to global resources.

To learn about regional vs. global resources and learn how to customize your AWS Config configuration, see Selecting which Resources AWS Config Records.

Viewing API Gateway configuration details in the AWS Config console

You can use the AWS Config console to look for API Gateway resources and get current and historical details about their configurations. The following procedure shows how to find information about an API Gateway API.

To find an API Gateway resource in the AWS config console

1. Open the AWS Config console.
2. Choose **Resources**.
3. On the **Resource** inventory page, choose **Resources**.
4. Open the **Resource type** menu, scroll to APIGateway or APIGatewayV2, and then choose one or more of the API Gateway resource types.
5. Choose **Look up**.
6. Choose a resource ID in the list of resources that AWS Config displays. AWS Config displays configuration details and other information about the resource you selected.
7. To see the full details of the recorded configuration, choose **View Details**.

To learn more ways to find a resource and view information on this page, see Viewing AWS Resource Configurations and History in the AWS Config Developer Guide.

Evaluating API Gateway resources using AWS Config rules

You can create AWS Config rules, which represent the ideal configuration settings for your API Gateway resources. You can use predefined AWS Config Managed Rules, or define custom rules. AWS Config continuously tracks changes to the configuration of your resources to determine whether those changes violate any of the conditions in your rules. The AWS Config console shows the compliance status of your rules and resources.

If a resource violates a rule and is flagged as noncompliant, AWS Config can alert you using an Amazon Simple Notification Service Developer Guide (Amazon SNS) topic. To programmatically consume the data in these AWS Config alerts, use an Amazon Simple Queue Service (Amazon SQS) queue as the notification endpoint for the Amazon SNS topic.

To learn more about setting up and using rules, see Evaluating Resources with Rules in the AWS Config Developer Guide.

Compliance validation for Amazon API Gateway

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. 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 baseline environments on AWS that are security and compliance focused.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map

the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).

- 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. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon API Gateway

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.

To prevent your APIs from being overwhelmed by too many requests, API Gateway throttles requests to your APIs. Specifically, API Gateway sets a limit on a steady-state rate and a burst of request submissions against all APIs in your account. You can configure custom throttling for your APIs. To learn more, see Throttle API requests for better throughput.

You can use Route 53 health checks to control DNS failover from an API Gateway API in a primary region to an API Gateway API in a secondary region. For an example, see the section called “DNS failover”.

Infrastructure security in Amazon API Gateway

As a managed service, Amazon API Gateway is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud

Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access API Gateway through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). 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.

You can call these API operations from any network location, but API Gateway does support resource-based access policies, which can include restrictions based on the source IP address. You can also use resource-based policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given API Gateway resource from only the specific VPC within the AWS network.

Vulnerability analysis in Amazon API Gateway

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model.

Security best practices in Amazon API Gateway

API Gateway 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.

Implement least privilege access

> Use IAM policies to implement least privilege access for creating, reading, updating, or deleting API Gateway APIs. To learn more, see Identity and access management for Amazon API Gateway. API Gateway offers several options to control access to APIs that you create. To learn

more, see Controlling and managing access to a REST API in API Gateway, Controlling and managing access to a WebSocket API in API Gateway, and Controlling access to HTTP APIs with JWT authorizers.

Implement logging

Use CloudWatch Logs or Amazon Data Firehose to log requests to your APIs. To learn more, see Monitoring REST APIs, Configuring logging for a WebSocket API, and Configuring logging for an HTTP API.

Implement Amazon CloudWatch alarms

Using CloudWatch alarms, you watch a single metric over a time period that you specify. If the metric exceeds a given threshold, a notification is sent to an Amazon Simple Notification Service topic or AWS Auto Scaling policy. CloudWatch alarms do not invoke actions when a metric is in a particular state. Rather, the state must have changed and been maintained for a specified number of periods. For more information, see the section called “CloudWatch metrics”.

Enable AWS CloudTrail

CloudTrail provides a record of actions taken by a user, role, or an AWS service in API Gateway. Using the information collected by CloudTrail, you can determine the request that was made to API Gateway, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, see the section called “Working with CloudTrail”.

Enable AWS Config

AWS Config provides a detailed view of the configuration of AWS resources in your account. You can see how resources are related, get a history of configuration changes, and see how relationships and configurations change over time. You can use AWS Config to define rules that evaluate resource configurations for data compliance. AWS Config rules represent the ideal configuration settings for your API Gateway resources. If a resource violates a rule and is flagged as noncompliant, AWS Config can alert you using an Amazon Simple Notification Service (Amazon SNS) topic. For details, see the section called “Working with AWS Config”.

Use AWS Security Hub

Monitor your usage of API Gateway as it relates to security best practices by using AWS Security Hub. Security Hub uses *security controls* to evaluate resource configurations and *security standards* to help you comply with various compliance frameworks. For more information about using Security Hub to evaluate API Gateway resources, see Amazon API Gateway controls in the *AWS Security Hub User Guide*.

Tagging your API Gateway resources

A *tag* is a metadata label that you assign or that AWS assigns to an AWS resource. Each tag has two parts:

- A *tag key* (for example, `CostCenter`, `Environment`, or `Project`). Tag keys are case sensitive.
- An optional field known as a *tag value* (for example, `111122223333` or `Production`). Omitting the tag value is the same as using an empty string. Like tag keys, tag values are case-sensitive.

Tags help you do the following:

- Control access to your resources based on the tags that are assigned to them. You control access by specifying tag keys and values in the conditions for an AWS Identity and Access Management (IAM) policy. For more information about tag-based access control, see Controlling Access Using Tags in the *IAM User Guide*.
- Track your AWS costs. You activate these tags on the AWS Billing and Cost Management dashboard. AWS uses the tags to categorize your costs and deliver a monthly cost allocation report to you. For more information, see Use Cost Allocation Tags in the AWS Billing User Guide.
- Identify and organize your AWS resources. Many AWS services support tagging, so you can assign the same tag to resources from different services to indicate that the resources are related. For example, you could assign the same tag to an API Gateway stage that you assign to a CloudWatch Events rule.

For tips on using tags, see the AWS Tagging Strategies post on the *AWS Answers* blog.

The following sections provide more information about tags for Amazon API Gateway.

Topics

API Gateway resources that can be tagged

Tags can be set on the following HTTP API or WebSocket API resources in the Amazon API Gateway V2 API:

- Api
- DomainName
- Stage
- VpcLink

In addition, tags can be set on the following REST API resources in the Amazon API Gateway V1 API:

- ApiKey
- ClientCertificate
- DomainName
- RestApi
- Stage
- UsagePlan
- VpcLink

Tags cannot be set directly on other resources. However, in the Amazon API Gateway V1 API, child resources inherit the tags that are set on parent resources. For example:

- If a tag is set on a RestApi resource, that tag is inherited by the following child resources of that RestApi for Attribute-based access control:
  - Authorizer
  - Deployment
  - Documentation
  - GatewayResponse
  - Integration
  - Method
  - Model
  - Resource
  - ResourcePolicy
  - Setting
  - Stage

- If a tag is set on a `DomainName`, that tag is inherited by any `BasePathMapping` resources under it.
- If a tag is set on a `UsagePlan`, that tag is inherited by any `UsagePlanKey` resources under it.

> **Note**
>
> Tag inheritance applies only to attribute-based access control. For example, you can't use inherited tags to monitor costs in AWS Cost Explorer. API Gateway doesn't return inherited tags when you call GetTags for a resource.

Tag inheritance in the Amazon API Gateway V1 API

Previously it was only possible to set tags on stages. Now that you can also set them on other resources, a `Stage` can receive a tag two ways:

- The tag can be set directly on the `Stage`.
- The stage can inherit the tag from its parent `RestApi`.

If a stage receives a tag both ways, the tag that was set directly on the stage takes precedence. For example, suppose a stage inherits the following tags from its parent REST API:

```
{
 'foo': 'bar',
 'x':'y'
}
```

Suppose it also has the following tags set on it directly:

```
{
 'foo': 'bar2',
 'hello': 'world'
}
```

The net effect would be for the stage to have the following tags, with the following values:

```
{
 'foo': 'bar2',
 'hello': 'world'
 'x':'y'
}
```

Tag restrictions and usage conventions

The following restrictions and usage conventions apply to using tags with API Gateway resources:

- Each resource can have a maximum of 50 tags.
- For each resource, each tag key must be unique, and each tag key can have only one value.
- The maximum tag key length is 128 Unicode characters in UTF-8.
- The maximum tag value length is 256 Unicode characters in UTF-8.
- Allowed characters for keys and values are letters, numbers, spaces representable in UTF-8, and the following characters: ***. : + = @ _ / -*** (hyphen). Amazon EC2 resources allow any characters.
- Tag keys and values are case-sensitive. As a best practice, decide on a strategy for capitalizing tags, and consistently implement that strategy across all resource types. For example, decide whether to use `Costcenter`, `costcenter`, or `CostCenter`, and use the same convention for all tags. Avoid using similar tags with inconsistent case treatment.
- The `aws:` prefix is prohibited for tags; it's reserved for AWS use. You can't edit or delete tag keys or values with this prefix. Tags with this prefix do not count against your tags per resource limit.

Using tags to control access to API Gateway REST API resources

Conditions in AWS Identity and Access Management policies are part of the syntax that you use to specify permissions to API Gateway resources. For details about specifying IAM policies, see the section called "Use IAM permissions". In API Gateway, resources can have tags, and some actions can include tags. When you create an IAM policy, you can use tag condition keys to control:

- Which users can perform actions on an API Gateway resource, based on tags that the resource already has.
- Which tags can be passed in an action's request.
- Whether specific tag keys can be used in a request.

Using tags for attribute-based access control can allow for finer control than API-level control, as well as more dynamic control than resource-based access control. IAM policies can be created that allow or disallow an operation based on tags provided in the request (request tags), or tags on the resource that is being operated on (resource tags). In general, resource tags are for resources that already exist. Request tags are for when you're creating new resources.

For the complete syntax and semantics of tag condition keys, see Controlling Access Using Tags in the *IAM User Guide*.

The following examples demonstrate how to specify tag conditions in policies for API Gateway users.

Limit actions based on resource tags

The following example policy grants users permission to perform all actions on all resources, as long as those resources don't have the tag `Environment` with a value of `prod`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "apigateway:*",
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Environment": "prod"
        }
      }
    }
  ]
}
```

Limit actions based on resource tags 1281

Allow actions based on resource tags

The following example policy allows users to perform all actions on API Gateway resources, as long as the resources have the tag `Environment` with a value of `Development`. The `Deny` statement prevents the user from changing the value of the `Environment` tag.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ConditionallyAllow",
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "arn:aws:apigateway:*::*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Environment": "Development"
        }
      }
    },
    {
      "Sid": "AllowTagging",
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*"
      ]
    },
    {
      "Sid": "DenyChangingTag",
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*"
      ],
```

Allow actions based on resource tags 1282

```
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:TagKeys": "Environment"
        }
      }
    }
  ]
}
```

Deny tagging operations

The following example policy allows a user to perform all API Gateway actions, except for changing tags.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "*"
      ],
    },
    {
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": "arn:aws:apigateway:*::/tags*",
    }
  ]
}
```

Allow tagging operations

The following example policy allows a user to get all API Gateway resources, and change tags for those resources. To get the tags for a resource, the user must have GET permissions for that resource. To update the tags for a resource, the user must have PATCH permissions for that resource.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*",
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:PATCH",
      ],
      "Resource": [
        "arn:aws:apigateway:*::*",
      ]
    }
  ]
}
```

API references

Amazon API Gateway provides APIs for creating and deploying your own HTTP and WebSocket APIs. In addition, API Gateway APIs are available in standard AWS SDKs.

If you are using a language for which an AWS SDK exists, you may prefer to use the SDK rather than using the API Gateway REST APIs directly. The SDKs make authentication simpler, integrate easily with your development environment, and provide easy access to API Gateway commands.

Here's where to find the AWS SDKs and API Gateway REST API reference documentation:

- Tools for Amazon Web Services
- Amazon API Gateway REST API Reference
- Amazon API Gateway WebSocket and HTTP API Reference

Amazon API Gateway quotas and important notes

Topics

Unless noted otherwise, the quotas can be increased upon request. To request a quota increase, you can use Service Quotas or contact the AWS Support Center.

When authorization is enabled on a method, the maximum length of the method's ARN (for example, `arn:aws:execute-api:{region-id}:{account-id}:{api-id}/{stage-id}/{method}/{resource}/{path}`) is 1600 bytes. The path parameter values (whose size is determined at runtime) can cause the ARN length to exceed the limit. When this happens, the API client receives a `414 Request URI too long` response.

> **Note**
>
> This limits URI length when resource policies are used. In the case of private APIs where a resource policy is required, this limits the URI length of all private APIs.

API Gateway account-level quotas, per Region

The following quotas apply per account, per Region in Amazon API Gateway.

Resource or operation	Default quota	Can be increased
Throttle quota per account, per Region	10,000 requests per second (RPS) with an additional burst capacity provided by the token bucket algorithm, using a maximum bucket capacity of 5,000 requests. *	Yes

Resource or operation	Default quota	Can be increased
across HTTP APIs, REST APIs, WebSocket APIs, and WebSocket callback APIs	**Note** The burst quota is determined by the API Gateway service team based on the overall RPS quota for the account in the Region. It is not a quota that a customer can control or request changes to.	
Regional APIs	600	No
Edge-optimized APIs	120	No

* For the Africa (Cape Town) and Europe (Milan) Regions, the default throttle quota is 2500 RPS and the default burst quota is 1250 RPS.

HTTP API quotas

The following quotas apply to configuring and running an HTTP API in API Gateway.

Resource or operation	Default quota	Can be increased
Routes per API	300	Yes
Integrations per API	300	No
Maximum integrati on timeout	30 seconds	No
Stages per API	10	Yes

Resource or operation	Default quota	Can be increased
Multi-level API mappings per domain	200	No
Tags per stage	50	No
Total combined size of request line and header values	10240 bytes	No
Payload size	10 MB	No
Custom domains per account per Region	120	Yes
Access log template size	3 KB	No
Amazon CloudWatch Logs log entry	1 MB	No
Authorizers per API	10	Yes
Audiences per authorizer	50	No
Scopes per route	10	No
Timeout for JSON Web Key Set endpoint	1500 ms	No

Resource or operation	Default quota	Can be increased
Response size from JSON Web Key Set endpoint	150000 bytes	No
Timeout for OpenID Connect discovery endpoint	1500 ms	No
Timeout for Lambda authorizer response	10000 ms	No
VPC links per account per Region	10	Yes
Subnets per VPC link	10	Yes
Stage variables per stage	100	No
Length, in characters, of the key in a stage variable	64	No
Length, in characters, of the value in a stage variable	512	No

API Gateway quotas for configuring and running a WebSocket API

The following quotas apply to configuring and running a WebSocket API in Amazon API Gateway.

Resource or operation	Default quota	Can be increased
New connections per second per account (across all WebSocket APIs) per Region	500	Yes
Concurrent connections	Not applicable *	Not applicable
AWS Lambda authorizers per API	10	Yes
AWS Lambda authorizer result size	8 KB	No
Routes per API	300	Yes
Integrations per API	300	Yes
Integration timeout	50 milliseconds - 29 seconds for all integration types, including Lambda, Lambda proxy, HTTP, HTTP proxy, and AWS integrations.	No
Stages per API	10	Yes
WebSocket frame size	32 KB	No

Resource or operation	Default quota	Can be increased
Message payload size	128 KB **	No
Connection duration for WebSocket API	2 hours	No
Idle Connection Timeout	10 minutes	No
Length, in characters, of the URL for a WebSocket API	4096	No

* API Gateway doesn't enforce a quota on concurrent connections. The maximum number of concurrent connections is determined by the rate of new connections per second and maximum connection duration of two hours. For example, with the default quota of 500 new connections per second, if clients connect at the maximum rate over two hours, API Gateway can serve up to 3,600,000 concurrent connections.

** Because of the WebSocket frame-size quota of 32 KB, a message larger than 32 KB must be split into multiple frames, each 32 KB or smaller. This applies to `@connections` commands. If a larger message (or larger frame size) is received, the connection is closed with code 1009.

API Gateway quotas for configuring and running a REST API

The following quotas apply to configuring and running a REST API in Amazon API Gateway.

Resource or operation	Default quota	Can be increased
Custom domain names per account per Region	120	Yes
Multi-level API mappings per domain	200	No
Length, in characters, of the URL for an edge-optimized API	8192	No
Length, in characters, of the URL for a regional API	10240	No
Private APIs per account per Region	600	No
Length, in characters, of API Gateway resource policy	8192	Yes
API keys per account per Region	10000	No
Client certificates per account per Region	60	Yes
Authorizers per API (AWS Lambda	10	Yes

Resource or operation	Default quota	Can be increased
and Amazon Cognito)		
Documentation parts per API	2000	Yes
Resources per API	300	Yes
Stages per API	10	Yes
Stage variables per stage	100	No
Length, in characters, of the key in a stage variable	64	No
Length, in characters, of the value in a stage variable	512	No
Usage plans per account per Region	300	Yes
Usage plans per API key	10	Yes
VPC links per account per Region	20	Yes

Resource or operation	Default quota	Can be increased
API caching TTL	300 seconds by default and configurable between 0 and 3600 by an API owner.	Not for the upper bound (3600)
Cached response size	1048576 Bytes. Cache data encryption may increase the size of the item that is being cached.	No
Integration timeout	50 milliseconds - 29 seconds for all integration types, including Lambda, Lambda proxy, HTTP, HTTP proxy, and AWS integrati ons.	Not for the lower or upper bounds.
Total combined size of all header values	10240 Bytes	No
Total combined size of all header values for a private API	8000 Bytes	No
Payload size	10 MB	No
Tags per stage	50	No
Number of iterations in a `#foreach ... #end` loop in mapping templates	1000	No

Resource or operation	Default quota	Can be increased
ARN length of a method with authorization	1600 bytes	No
Method-level throttling settings for a stage in a usage plan	20	Yes
Model size per API	400 KB	No
Number of certificates in a truststore	1,000 certificates up to 1 MB total object size.	No

For restapi:import or restapi:put, the maximum size of the API definition file is 6 MB.

All of the per-API quotas can only be increased on specific APIs.

API Gateway quotas for creating, deploying and managing an API

The following fixed quotas apply to creating, deploying, and managing an API in API Gateway, using the AWS CLI, the API Gateway console, or the API Gateway REST API and its SDKs. These quotas can't be increased.

Action	Default quota	Can be increased
CreateApiKey	5 requests per second per account	No
CreateDeployment	1 request every 5 seconds per account	No

Action	Default quota	Can be increased
CreateDocumentationVersion	1 request every 20 seconds per account	No
CreateDomainName	1 request every 30 seconds per account	No
CreateResource	5 requests per second per account	No
CreateRestApi	**Regional or private API** - 1 request every 3 seconds per account **Edge-optimized API** - 1 request every 30 seconds per account	No
CreateVpcLink (V2)	1 request every 15 seconds per account	No
DeleteApiKey	5 requests per second per account	No
DeleteDomainName	1 request every 30 seconds per account	No
DeleteResource	5 requests per second per account	No
DeleteRestApi	1 request every 30 seconds per account	No
GetResources	5 requests every 2 seconds per account	No

Action	Default quota	Can be increased
DeleteVpcLink (V2)	1 request every 30 seconds per account	No
ImportDocumentationParts	1 request every 30 seconds per account	No
ImportRestApi	**Regional or private API** - 1 request every 3 seconds per account **Edge-optimized API** - 1 request every 30 seconds per account	No
PutRestApi	1 request per second per account	No
UpdateAccount	1 request every 20 seconds per account	No
UpdateDomainName	1 request every 30 seconds per account	No
UpdateUsagePlan	1 request every 20 seconds per account	No
Other operations	No quota up to the total account quota.	No
Total operations	10 requests per second with a burst quota of 40 requests per second.	No

Amazon API Gateway important notes

Topics

Amazon API Gateway important notes for REST and WebSocket APIs

- API Gateway does not support sharing a custom domain name across REST and WebSocket APIs.
- Stage names can only contain alphanumeric characters, hyphens, and underscores. Maximum length is 128 characters.
- The `/ping` and `/sping` paths are reserved for the service health check. Use of these for API root-level resources with custom domains will fail to produce the expected result.
- API Gateway currently limits log events to 1024 bytes. Log events larger than 1024 bytes, such as request and response bodies, will be truncated by API Gateway before submission to CloudWatch Logs.
- CloudWatch Metrics currently limits dimension names and values to 255 valid XML characters. (For more information, see the CloudWatch User Guide.) Dimension values are a function of user-defined names, including API name, label (stage) name, and resource name. When choosing these names, be careful not to exceed CloudWatch Metrics limits.
- The maximum size of a mapping template is 300 KB.

Amazon API Gateway important notes for WebSocket APIs

- API Gateway supports message payloads up to 128 KB with a maximum frame size of 32 KB. If a message exceeds 32 KB, you must split it into multiple frames, each 32 KB or smaller. If a larger message is received, the connection is closed with code 1009.

Amazon API Gateway important notes for REST APIs

- The plain text pipe character (`|`) is not supported for any request URL query string and must be URL-encoded.

- The semicolon character (`;`) is not supported for any request URL query string and results in the data being split. In general, REST APIs decode URL-encoded request parameters before passing them to backend integrations.
- When using the API Gateway console to test an API, you may get an "unknown endpoint errors" response if a self-signed certificate is presented to the backend, the intermediate certificate is missing from the certificate chain, or any other unrecognizable certificate-related exceptions thrown by the backend.
- For an API `Resource` or `Method` entity with a private integration, you should delete it after removing any hard-coded reference of a `VpcLink`. Otherwise, you have a dangling integration and receive an error stating that the VPC link is still in use even when the `Resource` or `Method` entity is deleted. This behavior does not apply when the private integration references `VpcLink` through a stage variable.
- The following backends may not support SSL client authentication in a way that's compatible with API Gateway:
  - NGINX
  - Heroku
- API Gateway supports most of the OpenAPI 2.0 specification and the OpenAPI 3.0 specification, with the following exceptions:
  - Path segments can only contain alphanumeric characters, underscores, hyphens, periods, commas, colons, and curly braces. Path parameters must be separate path segments. For example, "resource/{path_parameter_name}" is valid; "resource{path_parameter_name}" is not.
  - Model names can only contain alphanumeric characters.
  - For input parameters, only the following attributes are supported: `name`, `in`, `required`, `type`, `description`. Other attributes are ignored.
  - The `securitySchemes` type, if used, must be `apiKey`. However, OAuth 2 and HTTP Basic authentication are supported via Lambda authorizers; the OpenAPI configuration is achieved via vendor extensions.
  - The `deprecated` field is not supported and is dropped in exported APIs.
  - API Gateway models are defined using JSON schema draft 4, instead of the JSON schema used by OpenAPI.
  - The `discriminator` parameter is not supported in any schema object.
  - The `example` tag is not supported.

- `exclusiveMinimum` is not supported by API Gateway.
- The `maxItems` and `minItems` tags are not included in simple request validation. To work around this, update the model after import before doing validation.
- `oneOf` is not supported for OpenAPI 2.0 or SDK generation.
- The `readOnly` field is not supported.
- `$ref` cannot be used to reference other files.
- Response definitions of the `"500": {"$ref": "#/responses/UnexpectedError"}` form is not supported in the OpenAPI document root. To work around this, replace the reference by the inline schema.
- Numbers of the `Int32` or `Int64` type are not supported. An example is shown as follows:

```
"elementId": {
    "description": "Working Element Id",
    "format": "int32",
    "type": "number"
}
```

- Decimal number format type (`"format": "decimal"`) is not supported in a schema definition.
- In method responses, schema definition must be of an object type and cannot be of primitive types. For example, `"schema": { "type": "string"}` is not supported. However, you can work around this using the following object type:

```
"schema": {
     "$ref": "#/definitions/StringResponse"
            }

 "definitions": {
    "StringResponse": {
      "type": "string"
    }
  }
```

- API Gateway doesn't use root level security defined in the OpenAPI specification. Hence security needs to be defined at an operation level to be appropriately applied.
- The `default` keyword is not supported.

- API Gateway enacts the following restrictions and limitations when handling methods with either Lambda integration or HTTP integration.
  - Header names and query parameters are processed in a case-sensitive way.
  - The following table lists the headers that may be dropped, remapped, or otherwise modified when sent to your integration endpoint or sent back by your integration endpoint. In this table:
    - `Remapped` means that the header name is changed from *`<string>`* to `X-Amzn-Remapped-`*`<string>`*.

      `Remapped Overwritten` means that the header name is changed from *`<string>`* to `X-Amzn-Remapped-`*`<string>`* and the value is overwritten.

Header name	**Request (`http/http_proxy /lambda`)**	**Response (`http/http_pr` `y /lambda`)**
`Age`	Passthrough	Passthrough
`Accept`	Passthrough	Dropped/Passthrough/Passthrough
`Accept-Charset`	Passthrough	Passthrough
`Accept-Encoding`	Passthrough	Passthrough
`Authorization`	Passthrough *	Remapped
`Connection`	Passthrough/Passthrough/Dropped	Remapped

Header name	Request (`http/http_proxy /lambda`)	Response (`http/http_pr y /lambda`)
`Content-Encoding`	Passthrough/Dropped/Passthrough	Passthrough
`Content-Length`	Passthrough (generated based on body)	Passthrough
`Content-MD5`	Dropped	Remapped
`Content-Type`	Passthrough	Passthrough
`Date`	Passthrough	Remapped Overwritten
`Expect`	Dropped	Dropped
`Host`	Overwritten to the integration endpoint	Dropped
`Max-Forwards`	Dropped	Remapped
`Pragma`	Passthrough	Passthrough
`Proxy-Authenticate`	Dropped	Dropped
`Range`	Passthrough	Passthrough
`Referer`	Passthrough	Passthrough
`Server`	Dropped	Remapped Overwritten

Header name	Request (`http/http_proxy /lambda`)	Response (`http/http_pr` `y /lambda`)
`TE`	Dropped	Dropped
`Transfer-Encoding`	Dropped/Dropped/Exception	Dropped
`Trailer`	Dropped	Dropped
`Upgrade`	Dropped	Dropped
`User-Agent`	Passthrough	Remapped
`Via`	Dropped/Dropped/Passthrough	Passthrough/Dropped/Dropped
`Warn`	Passthrough	Passthrough
`WWW-Authenticate`	Dropped	Remapped

* The `Authorization` header is dropped if it contains a Signature Version 4 signature or if `AWS_IAM` authorization is used.

- The Android SDK of an API generated by API Gateway uses the `java.net.HttpURLConnection` class. This class will throw an unhandled exception, on devices running Android 4.4 and earlier, for a 401 response resulted from remapping of the `WWW-Authenticate` header to `X-Amzn-Remapped-WWW-Authenticate`.
- Unlike API Gateway-generated Java, Android and iOS SDKs of an API, the JavaScript SDK of an API generated by API Gateway does not support retries for 500-level errors.
- The test invocation of a method uses the default content type of `application/json` and ignores specifications of any other content types.

- When sending requests to an API by passing the X-HTTP-Method-Override header, API Gateway overrides the method. So in order to pass the header to the backend, the header needs to be added to the integration request.
- When a request contains multiple media types in its Accept header, API Gateway only honors the first Accept media type. In the situation where you cannot control the order of the Accept media types and the media type of your binary content is not the first in the list, you can add the first Accept media type in the binaryMediaTypes list of your API, API Gateway will return your content as binary. For example, to send a JPEG file using an <img> element in a browser, the browser might send Accept:image/webp,image/*,*/*;q=0.8 in a request. By adding image/webp to the binaryMediaTypes list, the endpoint will receive the JPEG file as binary.
- Customizing the default gateway response for 413 REQUEST_TOO_LARGE isn't currently supported.
- API Gateway includes a Content-Type header for all integration responses. By default, the content type is application/json.

Document history

The following table describes the important changes to the documentation since the last release of Amazon API Gateway. For notification about updates to this documentation, you can subscribe to an RSS feed by choosing the RSS button in the top menu panel.

- **Latest documentation update:** February 15, 2024

Change	Description	Date
Added support for TLS 1.3	API Gateway now supports TLS 1.3 on Regional REST APIs, HTTP APIs, and WebSocket APIs. For more information, see Choosing a security policy for your custom domain in API Gateway.	February 15, 2024
REST API and WebSocket API console updates	Updated console information for REST APIs and WebSocket APIs	December 10, 2023
Documentation update	Updated conceptual information and created new tutorials for data transformations and request validation topics for API Gateway REST APIs. For more information, see Use request validation in API Gateway and Setting up data transformations for REST API.	June 22, 2023
Configure DNS failover for a multi-Region API Gateway	Added support to use Amazon Route 53 health checks to control DNS failover from an API Gateway REST API in a	October 31, 2022

	primary AWS Region to one in a secondary Region. For more information, see Configure custom health checks for DNS failover.	
Documentation update	Updated core feature summaries for REST API and HTTP API APIs. For more information, see Choosing between REST API and HTTP API APIs.	May 31, 2022
Managed policy update	Added `acm:GetCertificate` support to the `AWSServiceRoleForAPIGateway` policy. For more information, see Using service-linked roles for API Gateway.	July 12, 2021
Parameter mapping for HTTP APIs	Added support for parameter mapping for HTTP APIs. For more information, see Transforming API requests and responses.	January 7, 2021
Disable the default endpoint for a REST API	Added support for disabling the default endpoint for REST APIs. For more information, see Disabling the default endpoint for a REST API.	October 29, 2020

Mutual TLS authentication	Added support for mutual TLS authentication for REST APIs and HTTP APIs. For more information, see Configuring mutual TLS authentication for a REST API and Configuring mutual TLS authentication for an HTTP API.	September 17, 2020
HTTP API AWS Lambda authorizers	Added support for AWS Lambda authorizers for HTTP APIs. For more information, see Working with AWS Lambda authorizers for HTTP APIs.	September 9, 2020
HTTP API AWS service integrations	Added support for AWS service integrations for HTTP APIs. For more information, see Working with AWS service integrations for HTTP APIs.	August 20, 2020
HTTP API wildcard custom domains	Added support for wildcard custom domain names for HTTP APIs. For more information, see Wildcard custom domain names.	August 10, 2020
Serverless developer portal improvements	Added user management to the administrator panel and support for exporting API definitions. For more information, see Use the serverless developer portal to catalog your API Gateway APIs.	June 25, 2020

WebSocket API `Sec-WebSocket-Protocol` support	Added support for the `Sec-WebSocket-Protocol` field. For more information, see Setting up a $connect route that requires a WebSocket subprotocol.	June 16, 2020
HTTP API export	Added support for exporting OpenAPI 3.0 definitions of HTTP APIs. For more information, see Exporting an HTTP API from API Gateway.	April 20, 2020
Security documentation	Added security documentation. For more information, see Security in Amazon API Gateway.	March 31, 2020
Reorganized documentation	Reorganized the developer guide.	March 12, 2020
HTTP API general availability	Released HTTP APIs in general availability. For more information, see Working with HTTP APIs.	March 12, 2020
HTTP API logging	Added support for `$context.integrationErrorMessage` in HTTP API logs. For more information, see HTTP API Logging Variables.	February 26, 2020
AWS variables for OpenAPI import	Added support for AWS variables in OpenAPI definitions. For more information, see AWS Variables for OpenAPI Import.	February 17, 2020

HTTP APIs	Released HTTP APIs in beta. For more information, see HTTP APIs.	December 4, 2019
Wildcard custom domain names	Added support for wildcard custom domain names. For more information, see Wildcard Custom Domain Names.	October 21, 2019
Amazon Data Firehose logging	Added support for Amazon Data Firehose as a destination for access logging data. For more information, see Using Amazon Data Firehose as a Destination for API Gateway Access Logging.	October 15, 2019
Route53 aliases for invoking private APIs	Added support for additiona l Route53 alias DNS records for invoking private APIs. For more information, see Accessing Your Private API Using Route53 Alias.	September 18, 2019
Tag-based access control for WebSocket APIs	Added support for tag-based access control for WebSocket APIs. For more information, see API Gateway Resources That Can Be Tagged.	June 27, 2019

TLS version selection for custom domains	Added support for Transport Layer Security (TLS) version selection for APIs that are deployed to custom domains. See the note in Choose a Minimum TLS Version for a Custom Domain in API Gateway.	June 20, 2019
VPC endpoint policies for private APIs	Added support for improving the security of private APIs by attaching endpoint policies to interface VPC endpoints . For more information, see Use VPC Endpoint Policies for Private APIs in API Gateway.	June 4, 2019
Documentation updated	Rewrote Getting Started with Amazon API Gateway. Moved tutorials to Amazon API Gateway Tutorials.	May 29, 2019
Tag-based access control for REST APIs	Added support for tag-based access control for REST APIs. For more information, see Using Tags with IAM Policies to Control Access to API Gateway Resources.	May 23, 2019

Documentation updated	Rewrote 6 topics: What Is Amazon API Gateway?, Tutorial: Build an API with HTTP Proxy Integration, Tutorial: Create a Calc REST API with Three Non-Proxy Integrations, API Gateway Mapping Template and Access Logging Variable Reference , Use API Gateway Lambda Authorizers, and Enable CORS for an API Gateway REST API Resource.	April 5, 2019
Serverless developer portal improvements	Added administrator panel and other improvements to make it easier to publish APIs in the Amazon API Gateway developer portal. For more information, see Use a Developer Portal to Catalog Your APIs.	March 28, 2019
Support for AWS Config	Added support for AWS Config. For more information, see Monitoring API Gateway API Configuration with AWS Config.	March 20, 2019
Support for AWS CloudFormation	Added API Gateway V2 API to the AWS CloudFormation template reference. For more information, see Amazon API Gateway V2 Resource Types Reference.	February 7, 2019

Support for WebSocket APIs	Added support for WebSocket APIs. For more information, see Creating a WebSocket API in Amazon API Gateway.	December 18, 2018
Serverless developer portal available through AWS Serverless Application Repository	The Amazon API Gateway developer portal serverless application is now available from the AWS Serverless Application Repository (in addition to GitHub). For more information, see Use a Developer Portal to Catalog Your API Gateway APIs.	November 16, 2018
Support for AWS WAF	Added support for AWS WAF (Web Application Firewall) . For more information, see Control Access to an API Using AWS WAF.	November 5, 2018
Serverless developer portal	Amazon API Gateway now provides a fully customizable developer portal as a serverless application that you can deploy for publishing your API Gateway APIs. For more information, see Use a Developer Portal to Catalog Your API Gateway APIs.	October 29, 2018

Support for multi-value headers and query string parameters	Amazon API Gateway now supports multiple headers and query string parameters that have the same name. For more information, see Support for Multi-Value Headers and Query String Parameters.	October 4, 2018
OpenAPI support	Amazon API Gateway now supports OpenAPI 3.0 as well as OpenAPI (Swagger) 2.0.	September 27, 2018
Documentation updated	Added a new topic: How Amazon API Gateway Resource Policies Affect Authorization Workflow.	September 27, 2018
Active AWS X-Ray integration	You can now use AWS X-Ray to trace and analyze latencies in user requests as they travel through your APIs to the underlying services. For more information, see Trace API Gateway API Execution with AWS X-Ray.	September 6, 2018

Caching improvements	Only GET methods will have caching enabled by default when you enable caching for an API stage. This helps to ensure the safety of your API. You can enable caching for other methods by overriding method settings. For more information, see Enable API Caching to Enhance Responsiveness.	August 20, 2018
Service limits revised	Several limits have been revised: Increased number of APIs per account. Increased API rate limits for Create/Import/Deploy APIs. Corrected some rates from per minute to per second. For more information, see Limits.	July 13, 2018
Overriding API request and response parameters and headers	Added support for overriding request headers, query strings, and paths, as well as response headers and status codes. For more information, see Use a Mapping Template to Override an API's Request and Response Parameters and Headers.	July 12, 2018

Method-level throttling for usage plans	Added support for setting default per-method throttling limits, as well as setting throttling limits for individual API methods in usage plan settings. These settings are in addition to the existing account-level throttling and default method-level throttling limits that you can set in stage settings. For more information, see Throttle API Requests for Better Throughput.	July 11, 2018
API Gateway Developer Guide update notifications now available through RSS	The HTML version of the API Gateway Developer Guide now supports an RSS feed of updates that are documented on this **Document History** page. The RSS feed includes updates made June 27, 2018, and later. Previously announced updates are still available on this page. Use the RSS button in the top menu panel to subscribe to the feed.	June 27, 2018

Earlier updates

The following table describes important changes in each release of the *API Gateway Developer Guide* before June 27, 2018.

Change	Description	Date changed
Private APIs	Added support for private APIs, which you expose via interface VPC endpoints. Traffic to your private APIs does not leave the Amazon network; it is isolated from the public internet.	June 14, 2018
Cross-Account Lambda Authorizers and Integrations and Cross-Account Amazon Cognito User Pool Authorizers	Use an AWS Lambda function from a different AWS account as a Lambda authorizer function or as an API integration backend. Or use an Amazon Cognito user pool as an authorizer. The other account can be in any region where Amazon API Gateway is available. For more information, see the section called "Configure a cross-account Lambda authorizer", the section called "Tutorial : Build an API with cross-account Lambda proxy integration", and the section called "Configure cross-account Amazon Cognito authorizer for a REST API".	April 2, 2018
Resource Policies for APIs	Use API Gateway resource policies to enable users from a different AWS account to securely access your API or to allow the API to be invoked only from specified source IP address ranges or CIDR blocks. For more information, see the section called "Use API Gateway resource policies".	April 2, 2018
Tagging for API Gateway resources	Tag an API stage with up to 50 tags for cost allocation of API requests and caching in API Gateway. For more information see the section called "Set up tags".	December 19, 2017
Payload compression and decompression	Enable calling your API with compressed payloads using one of the supported content codings. The compressed payloads are subject to mapping if a body-mapping template is specified. For more information, see the section called "Content encoding".	December 19, 2017
API key sourced from a custom authorizer	Return an API key from a custom authorizer to API Gateway to apply a usage plan for API methods that	December 19, 2017

Change	Description	Date changed
	require the key. For more information, see the section called "Choose an API key source".	
Authorization with OAuth 2 scopes	Enable authorization of method invocation by using OAuth 2 scopes with the `COGNITO_USER_POOLS` authorizer. For more information, see the section called "Use Amazon Cognito user pool as authorizer for a REST API".	December 14, 2017
Private Integration and VPC Link	Create an API with the API Gateway private integration to provide clients with access to HTTP/HTTPS resources in an Amazon VPC from outside of the VPC through a VpcLink resource. For more information, see the section called "Tutorial: Build an API with private integration" and the section called "Private integration".	November 30, 2017
Deploy a Canary for API testing	Add a canary release to an existing API deployment to test a newer version of the API while keeping the current version in operation on the same stage. You can set a percentage of stage traffic for the canary release and enable canary-specific execution and access logged in separate CloudWatch Logs logs. For more information, see the section called "Set up a canary release deploymen t".	November 28, 2017
Access Logging	Log client access to your API with data derived from $context variables in a format of your choosing. For more information, see the section called "CloudWatch logs".	November 21, 2017
Ruby SDK of an API	Generate a Ruby SDK for your API and use it to invoke your API methods. For more information, see the section called "Generate the Ruby SDK of an API" and the section called "Use a Ruby SDK generated by API Gateway for a REST API".	November 20, 2017

Change	Description	Date changed
Regional API endpoint	Specify a regional API endpoint to create an API for non-mobile clients. A non-mobile client, such as an EC2 instance, runs in the same AWS Region where the API is deployed. As with an edge-optimized API, you can create a custom domain name for a regional API. For more information, see the section called "Set up a Regional API" and the section called "Setting up a regional custom domain name".	November 2, 2017
Custom request authorizer	Use custom request authorizer to supply user-auth enticating information in request parameters to authorize API method calls. The request parameters include headers and query string parameters as well as stage and context variables. For more information, see Use API Gateway Lambda authorizers.	September 15, 2017
Customizing gateway responses	Customize API Gateway-generated gateway responses to API requests that failed to reach the integration backend. A customized gateway message can provide the caller with API-specific custom error messages, including returning needed CORS headers, or can transform the gateway response data to a format of an external exchange. For more information, see Setting up gateway responses to customize error responses.	June 6, 2017
Mapping Lambda custom error properties to method response headers	Map individual custom error properties returned from Lambda to the method response header parameters using the `integration.response.body` parameter , relying API Gateway to deserialize the stringified custom error object at run time. For more information, see Handle custom Lambda errors in API Gateway.	June 6, 2017

Change	Description	Date changed
Throttling limits increase	Increase the account-level steady-state request rate limit to 10,000 requests per second (rps) and the bust limit to 5000 concurrent requests. For more information, see Throttle API requests for better throughput.	June 6, 2017
Validating method requests	Configure basic request validators on the API level or method levels so that API Gateway can validate incoming requests. API Gateway verifies that required parameter s are set and not blank, and verifies that the format of applicable payloads conforms to the configured model. For more information, see Use request validation in API Gateway.	April 11, 2017
Integrating with ACM	Use ACM Certificates for your API's custom domain names. You can create a certificate in AWS Certificate Manager or import an existing PEM-formatted certifica te into ACM. You then refer to the certificate's ARN when setting a custom domain name for your APIs. For more information, see Setting up custom domain names for REST APIs.	March 9, 2017
Generating and calling a Java SDK of an API	Let API Gateway generate the Java SDK for your API and use the SDK to call the API in your Java client. For more information, see Use a Java SDK generated by API Gateway for a REST API.	January 13, 2017
Integrating with AWS Marketplace	Sell your API in a usage plan as a SaaS product through AWS Marketplace. Use AWS Marketplace to extend the reach of your API. Rely on AWS Marketplace for customer billing on your behalf. Let API Gateway handle user authorization and usage metering. For more information, see Sell your APIs as SaaS.	December 1, 2016

Change	Description	Date changed
Enabling Documentation Support for your API	Add documentation for API entities in DocumentationPart resources in API Gateway. Associate a snapshot of the collection `DocumentationPart` instances with an API stage to create a DocumentationVersion. Publish API documentation by exporting a documentation version to an external file, such as a Swagger file. For more information, see Documenting REST APIs.	December 1, 2016
Updated custom authorizer	A customer authorizer Lambda function now returns the caller's principal identifier. The function also can return other information as key-value pairs of the `context` map and an IAM policy. For more information, see Output from an Amazon API Gateway Lambda authorizer.	December 1, 2016
Supporting binary payloads	Set binaryMediaTypes on your API to support binary payloads of a request or response. Set the `contentHandling` property on an Integration or IntegrationResponse to specify whether to handle a binary payload as the native binary blob, as a Base64-enocded string, or as a passthrough without modifications. For more information, see Working with binary media types for REST APIs.	November 17, 2016
Enabling a proxy integration with an HTTP or Lambda backend through a proxy resource of an API.	Create a proxy resource with a greedy path parameter of the form `{proxy+}` and the catch-all `ANY` method. The proxy resource is integrated with an HTTP or Lambda backend using the HTTP or Lambda proxy integration, respectively. For more information, see Set up a proxy integration with a proxy resource.	September 20, 2016

Change	Description	Date changed
Extending selected APIs in API Gateway as product offerings for your customers by providing one or more usage plans.	Create a usage plan in API Gateway to enable selected API clients to access specified API stages at agreed-up on request rates and quotas. For more information, see Creating and using usage plans with API keys.	August 11, 2016
Enabling method-level authorization with a user pool in Amazon Cognito	Create a user pool in Amazon Cognito and use it as your own identity provider. You can configure the user pool as a method-level authorizer to grant access for users who are registered with the user pool. For more information, see Control access to a REST API using Amazon Cognito user pools as authorizer.	July 28, 2016
Enabling Amazon CloudWatch metrics and dimensions under the `AWS/ApiGateway` namespace.	The API Gateway metrics are now standardized under the CloudWatch namespace of `AWS/ApiGateway` . You can view them in both the API Gateway console and the Amazon CloudWatch console. For more information, see Amazon API Gateway dimensions and metrics.	July 28, 2016
Enabling certifica te rotation for a custom domain name	Certificate rotation allows you to upload and renew an expiring certificate for a custom domain name. For more information, see Rotate a certificate imported into ACM.	April 27, 2016
Documenting changes for the updated Amazon API Gateway console.	Learn how to create and set up an API using the updated API Gateway console. For more information, see Tutorial: Create a REST API by importing an example and Tutorial: Build a REST API with HTTP non-proxy integration.	April 5, 2016

Change	Description	Date changed
Enabling the Import API feature to create a new or update an existing API from external API definitions.	With the Import API features, you can create a new API or update an existing one by uploading an external API definition expressed in Swagger 2.0 with the API Gateway extensions. For more information about the Import API, see Configuring a REST API using OpenAPI.	April 5, 2016
Exposing the `$input.body` variable to access the raw payload as string and the `$util.parseJson()` function to turn a JSON string into a JSON object in a mapping template.	For more information about `$input.body` and `$util.parseJson()` , see API Gateway mapping template and access logging variable reference.	April 5, 2016
Enabling client requests with method-level cache invalidation, and improving request throttling management.	Flush API stage-level cache and invalidate individual cache entry. For more information, see Flush the API stage cache in API Gateway and Invalidate an API Gateway cache entry. Improve the console experience for managing API request throttling. For more information, see Throttle API requests for better throughput.	March 25, 2016
Enabling and calling API Gateway API using custom authorization	Create and configure an AWS Lambda function to implement custom authorization. The function returns an IAM policy document that grants the Allow or Deny permissions to client requests of an API Gateway API. For more information, see Use API Gateway Lambda authorizers.	February 11, 2016

Change	Description	Date changed
Importing and exporting API Gateway API using a Swagger definition file and extensions	Create and update your API Gateway API using the Swagger specification with the API Gateway extensions. Import the Swagger definitions using the API Gateway Importer. Export an API Gateway API to a Swagger definition file using the API Gateway console or API Gateway Export API. For more information, see Configuring a REST API using OpenAPI and Export a REST API from API Gateway.	December 18, 2015
Mapping request or response body or body's JSON fields to request or response parameters.	Map method request body or its JSON fields into integration request's path, query string, or headers. Map integration response body or its JSON fields into request response's headers. For more information, see Amazon API Gateway API request and response data mapping reference.	December 18, 2015
Working with Stage Variables in Amazon API Gateway	Learn how to associate configuration attributes with a deployment stage of an API in Amazon API Gateway. For more information, see Setting up stage variables for a REST API deployment.	November 5, 2015
How to: Enable CORS for a Method	It is now easier to enable cross-origin resource sharing (CORS) for methods in Amazon API Gateway. For more information, see Enabling CORS for a REST API resource.	November 3, 2015
How to: Use Client Side SSL Authentication	Use Amazon API Gateway to generate SSL certificates that you can use to authenticate calls to your HTTP backend. For more information, see Generate and configure an SSL certificate for backend authentication.	September 22, 2015
Mock integration of methods	Learn how to mock-integrate an API with Amazon API Gateway. This feature enables developers to generate API responses from API Gateway directly without the need for a final integration backend beforehand.	September 1, 2015

Change	Description	Date changed
Amazon Cognito Identity support	Amazon API Gateway has expanded the scope of the `$context` variable so that it now returns information about Amazon Cognito Identity when requests are signed with Amazon Cognito credentials. In addition, we have added a `$util` variable for escaping characters in JavaScript and encoding URLs and strings. For more information, see API Gateway mapping template and access logging variable reference.	August 28, 2015
Swagger integration	Use the Swagger import tool on GitHub to import Swagger API definitions into Amazon API Gateway. Learn more about Working with API Gateway extensions to OpenAPI to create and deploy APIs and methods using the import tool. With the Swagger importer tool you can also update existing APIs.	July 21, 2015
Mapping Template Reference	Read about the `$input` parameter and its functions in the API Gateway mapping template and access logging variable reference.	July 18, 2015
Initial public release	This is the initial public release of the *API Gateway Developer Guide*.	July 9, 2015

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.