
Amazon API Gateway

Developer Guide



Amazon API Gateway: Developer Guide

Copyright © 2018 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

What Is Amazon API Gateway?	1
Gateway to AWS Cloud and Beyond	1
Part of AWS Serverless Infrastructure	2
Developer Experiences	2
Creating and Managing an API Gateway API	2
Calling an API Gateway API	3
Benefits of API Gateway	3
API Gateway Concepts	3
Getting Started	6
Get Ready to Build API Gateway API	6
Sign up for an AWS Account	6
Create IAM Users, Groups, Roles, and Policies in Your AWS Account	7
Create IAM Policies to Configure API Gateway Resources and to Call a Deployed API	7
Next Step	9
Build an API from an Example	9
Create and Test an Example API in the Console	10
Next Step	17
See Also	18
Build an API with Lambda Integration	18
Build an API with Lambda Proxy Integration	19
Build an API with Cross-Account Lambda Proxy Integration	27
Build an API with Lambda Custom Integration	29
Build an API with HTTP Integrations	38
Build an API with HTTP Proxy Integration	39
Build an API with HTTP Custom Integration	44
Build an API with Private Integration	74
Build an API with AWS Integration	75
Prerequisites	76
Step 1: Create the Resource	76
Step 2: Create the GET Method	77
Step 3: Create the AWS Service Proxy Execution Role	77
Step 4: Specify Method Settings and Test the Method	78
Step 5: Deploy the API	79
Step 6: Test the API	79
Step 7: Clean Up	80
Creating an API	81
Choose an Endpoint Type to Set up an API	81
Initialize API Setup	82
Set up an API Using the API Gateway Console	82
Set up an Edge-Optimized API Using AWS CLI Commands	83
Set up an Edge-Optimized API Using the AWS SDK for Node.js	87
Set up an Edge-Optimized API Using the API Gateway REST API	94
Set up an Edge-Optimized API by Importing Swagger Definitions	101
Set up a Regional API	103
Set up API Methods	106
Set up Method Request	107
Set up Method Response	113
Set up Method Using the Console	115
Set up API Integrations	118
Set up Integration Request	119
Set up Integration Response	125
Set up Lambda Integrations	126
Set up HTTP Integrations	142
Set up Private Integrations	146

Set up Mock Integrations	152
Set up Gateway Responses	155
Gateway Responses in API Gateway	156
Gateway Response Types	156
Set up a Gateway Response Using the API Gateway Console	159
Set up a Gateway Response Using the API Gateway REST API	161
Set up Gateway Response Customization in Swagger	161
Set up Data Mappings	162
Set up Request and Response Data Mappings Using the Console	162
Create Models and Mapping Templates	164
Request and Response Data Mapping Reference	187
Mapping Template Reference	191
Support Binary Payloads	199
Content Type Conversions in API Gateway	200
Enable Binary Support Using the API Gateway Console	202
Enable Binary Support Using API Gateway REST API	206
Import and Export Content Encodings	210
Examples of Binary Support	210
Enable Payload Compression	217
Enable Compression for an API	217
Call a Method with a Compressed Payload	220
Receive a Response with a Compressed Payload	220
Enable Request Validation	221
Overview of Basic Request Validation in API Gateway	222
Set up Basic Request Validation in API Gateway	222
Test Basic Request Validation in API Gateway	227
Swagger Definitions of a Sample API with Basic Request Validation	230
Update and Maintain an API	233
Update an API Endpoint Type	234
Maintain an API Using the Console	236
Import an API	238
Import an Edge-Optimized API	238
Import a Regional API	240
Import a Swagger File to Update an Existing API Definition	240
Set the Swagger basePath Property	242
Errors and Warnings during Import	243
Controlling Access to an API	245
Use API Gateway Resource Policies	245
Access Policy Language Overview for Amazon API Gateway	246
API Gateway Resource Policy Examples	247
Create and Attach an API Gateway Resource Policy to an API	248
Interactions Between API Gateway Resource Policies and IAM Policies	250
AWS Conditions that can be used in API Gateway Resource Policies	251
Use IAM Permissions	252
API Gateway Permissions Model for Creating and Managing an API	252
API Gateway Permissions Model for Invoking an API	252
Control Access for Managing an API	254
Control Cross-Account Access to Your API	256
Control Access for Invoking an API	258
IAM Policy Examples for Managing API Gateway APIs	261
IAM Policy Examples for API Execution Permissions	265
Create and Attach a Policy to an IAM User	266
Enable CORS for a Resource	267
Prerequisites	268
Enable CORS Using the Console	268
Enable CORS Using Swagger Definition	270
Use Lambda Authorizers	272

Types of API Gateway Lambda Authorizers	273
Create a Lambda Authorizer Lambda Function	273
Input to a Lambda Authorizer	277
Output from an Amazon API Gateway Lambda Authorizer	279
Configure Lambda Authorizer	280
Call an API with Lambda Authorizers	282
Configure Cross-Account Lambda Authorizer	285
Use Amazon Cognito User Pools	286
Obtain Permissions to Create User Pool Authorizers	287
Create an Amazon Cognito User Pool	288
Integrate an API with a User Pool	289
Call an API Integrated with a User Pool	293
Use Client-Side SSL Certificates	293
Generate a Client Certificate Using the API Gateway Console	294
Configure an API to Use SSL Certificates	294
Test Invoke to Verify the Client Certificate Configuration	295
Configure Backend HTTPS Server to Verify the Client Certificate	297
Rotate an Expiring Client Certificate	297
Supported Certificate Authorities for HTTP and HTTP Proxy Integration	298
Use API Gateway Usage Plans	314
What Is a Usage Plan?	315
Expose API Key Source	315
How to Configure a Usage Plan?	317
Set Up API Keys Using the API Gateway Console	317
Create, Configure, and Test Usage Plans with the API Gateway Console	320
Set Up API Keys Using the API Gateway REST API	324
Create, Configure, and Test Usage Plans Using the API Gateway REST API	325
API Gateway API Key File Format	327
Documenting an API	329
Representation of API Documentation in API Gateway	329
Documentation Parts	329
Documentation Versions	336
Document an API Using the API Gateway Console	337
Document the API Entity	337
Document a RESOURCE Entity	340
Document a METHOD Entity	340
Document a QUERY_PARAMETER Entity	341
Document a PATH_PARAMETER Entity	342
Document a REQUEST_HEADER Entity	342
Document a REQUEST_BODY Entity	343
Document a RESPONSE Entity	343
Document a RESPONSE_HEADER Entity	343
Document a RESPONSE_BODY Entity	344
Document a MODEL Entity	344
Document an AUTHORIZER Entity	345
Document an API Using the API Gateway REST API	345
Document the API Entity	346
Document a RESOURCE Entity	347
Document a METHOD Entity	349
Document a QUERY_PARAMETER Entity	352
Document a PATH_PARAMETER Entity	353
Document a REQUEST_BODY Entity	353
Document a REQUEST_HEADER Entity	354
Document a RESPONSE Entity	355
Document a RESPONSE_HEADER Entity	356
Document an AUTHORIZER Entity	357
Document a MODEL Entity	358

Update Documentation Parts	360
List Documentation Parts	360
Publish API Documentation	360
Create a Documentation Snapshot and Associate it with an API Stage	361
Create a Documentation Snapshot	361
Update a Documentation Snapshot	362
Get a Documentation Snapshot	362
Associate a Documentation Snapshot with an API Stage	362
Download a Documentation Snapshot Associated with a Stage	363
Import API Documentation	366
Importing Documentation Parts Using the API Gateway REST API	366
Importing Documentation Parts Using the API Gateway Console	368
Control Access to API Documentation	368
Deploying an API	370
Create a Deployment	371
Create a Deployment Using AWS CLI	371
Deploy API from the Console	372
Set up a Stage	373
Set up a Stage Using the API Gateway Console	373
Throttle API Requests	376
Enable API Caching	378
Set up API Logging	382
Set up Stage Variables	385
Set Up Tags for an API Stage	395
Set up a Canary Release Deployment	397
Canary Release Deployment in API Gateway	398
Create a Canary Release Deployment	399
Update a Canary Release	406
Promote a Canary Release	408
Disable a Canary Release	412
Export an API	414
Request to Export an API	414
Download API Swagger Definition in JSON	414
Download API Swagger Definition in YAML	415
Download API Swagger Definition with Postman Extensions in JSON	415
Download API Swagger Definition with API Gateway Integration in YAML	415
Export API Using the API Gateway Console	415
Generate SDK of an API	416
Generate SDKs for an API Using the API Gateway Console	416
Generate SDKs for an API Using AWS CLI Commands	419
Simple Calculator Lambda Function	420
Simple Calculator API in API Gateway	422
Simple Calculator API Swagger Definition	427
Set up an API Custom Domain Name	432
Get Certificates Ready in AWS Certificate Manager	434
How to Create an Edge-Optimized Custom Domain Name	436
Set up a Regional Custom Domain Name	442
Migrate Custom Domain Names	447
Sell Your API as SaaS	453
Initialize AWS Marketplace Integration with API Gateway	453
Handle Customer Subscription to Usage Plans	455
Invoking an API	457
Obtain an API's Invoke URL in the API Gateway Console	457
Use the Console to Test a Method	458
Prerequisites	458
Test a Method with the API Gateway Console	458
Use Postman to Call an API	459

Call API through Generated SDKs	459
Use a Java SDK Generated by API Gateway	460
Use an Android SDK Generated by API Gateway	463
Use a JavaScript SDK Generated by API Gateway	465
Use a Ruby SDK Generated by API Gateway	466
Use iOS SDK Generated by API Gateway in Objective-C or Swift	469
Call API through AWS Amplify JavaScript Library	477
Trace API Management and Invocation	477
Log API Management Calls with CloudTrail	478
Monitor API execution with Amazon CloudWatch	479
Swagger Extensions	486
x-amazon-apigateway-any-method	486
x-amazon-apigateway-any-method Example	487
x-amazon-apigateway-api-key-source	487
x-amazon-apigateway-api-key-source Example	487
x-amazon-apigateway-authorizer	488
x-amazon-apigateway-authorizer Examples	489
x-amazon-apigateway-auth-type	491
x-amazon-apigateway-auth-type Example	491
See Also	491
x-amazon-apigateway-binary-media-type	492
x-amazon-apigateway-binary-media-types Example	492
x-amazon-apigateway-documentation	492
x-amazon-apigateway-documentation Example	492
x-amazon-apigateway-gateway-responses	493
x-amazon-apigateway-gateway-responses Example	493
x-amazon-apigateway-gateway-responses.gatewayResponse	493
x-amazon-apigateway-gateway-responses.gatewayResponse Example	494
x-amazon-apigateway-gateway-responses.responseParameters	494
x-amazon-apigateway-gateway-responses.responseParameters Example	495
x-amazon-apigateway-gateway-responses.responseTemplates	495
x-amazon-apigateway-gateway-responses.responseTemplates Example	495
x-amazon-apigateway-integration	496
x-amazon-apigateway-integration Example	498
x-amazon-apigateway-integration.requestTemplates	499
x-amazon-apigateway-integration.requestTemplates Example	500
x-amazon-apigateway-integration.requestParameters	500
x-amazon-apigateway-integration.requestParameters Example	500
x-amazon-apigateway-integration.responses	501
x-amazon-apigateway-integration.responses Example	501
x-amazon-apigateway-integration.response	502
x-amazon-apigateway-integration.response Example	503
x-amazon-apigateway-integration.responseTemplates	503
x-amazon-apigateway-integration.responseTemplate Example	503
x-amazon-apigateway-integration.responseParameters	504
x-amazon-apigateway-integration.responseParameters Example	504
x-amazon-apigateway-request-validator	504
x-amazon-apigateway-request-validator Example	504
x-amazon-apigateway-requestValidators	505
x-amazon-apigateway-requestValidators Example	505
x-amazon-apigateway-requestValidators.requestValidator	506
x-amazon-apigateway-requestValidators.requestValidator Example	506
Samples and Tutorials	507
Create an API for Lambda Functions	507
Set Up an IAM Role and Policy for an API to Invoke Lambda Functions	509
Create a Lambda Function in the Backend	509
Create API Resources for the Lambda Function	511

Create a GET Method with Query Parameters to Call the Lambda Function	511
Create a POST Method with a JSON Payload to Call the Lambda Function	515
Create a GET Method with Path Parameters to Call the Lambda Function	517
Swagger Definitions of a Sample API for a Lambda Function	521
Create an API as an Amazon S3 Proxy	526
Set Up IAM Permissions for the API to Invoke Amazon S3 Actions	526
Create API Resources to Represent Amazon S3 Resources	528
Expose an API Method to List the Caller's Amazon S3 Buckets	529
Expose API Methods to Access an Amazon S3 Bucket	534
Expose API Methods to Access an Amazon S3 Object in a Bucket	537
Call the API Using a REST API Client	540
Swagger Definitions of a Sample API as an Amazon S3 Proxy	542
Create an API as an Amazon Kinesis Proxy	551
Create an IAM Role and Policy for the API to Access Kinesis	552
Start to Create an API as a Kinesis Proxy	553
List Streams in Kinesis	554
Create, Describe, and Delete a Stream in Kinesis	558
Get Records from and Add Records to a Stream in Kinesis	563
Swagger Definitions of a Sample API as a Kinesis Proxy	574
API Gateway REST API	581
Limits, Pricing and Known Issues	582
API Gateway Limits	582
API Gateway Limits for Configuring and Running an API	582
API Gateway Limits for Creating, Deploying and Managing an API	583
API Gateway Pricing	584
Known Issues	585
Document History	588
AWS Glossary	593

What Is Amazon API Gateway?

Amazon API Gateway is an AWS service that enables developers to create, publish, maintain, monitor, and secure APIs at any scale. You can create APIs that access AWS or other web services, as well as data stored in the [AWS Cloud](#).

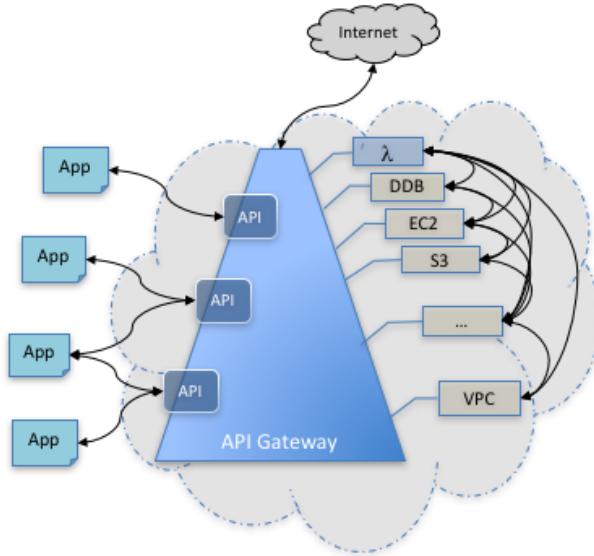
Topics

- [Gateway to AWS Cloud and Beyond \(p. 1\)](#)
- [Developer Experiences \(p. 2\)](#)
- [Benefits of API Gateway \(p. 3\)](#)
- [Amazon API Gateway Concepts \(p. 3\)](#)

Gateway to AWS Cloud and Beyond

API Gateway can be considered a [backplane](#) in the cloud to connect [AWS services](#) and other public or private websites. It provides consistent RESTful application programming interfaces (APIs) for mobile and web applications to access AWS services.

The following diagram shows API Gateway architecture.



In practical terms, API Gateway lets you create, configure, and host a [RESTful API](#) to enable applications to access the AWS Cloud. For example, an application can call an API in API Gateway to upload a user's annual income and expense data to [Amazon Simple Storage Service](#) or [Amazon DynamoDB](#), process the data in [AWS Lambda](#) to compute tax owed, and file a tax return via the IRS website.

As shown in the diagram, an app (or client application) gains programmatic access to AWS services, or a website on the internet, through one or more APIs, which are hosted in API Gateway. The app is at

the API's frontend. The integrated AWS services and websites are located at the API's backend. In API Gateway, the frontend is encapsulated by [method requests](#) and [method responses](#), and the backend is encapsulated by [integration requests](#) and [integration responses](#).

With Amazon API Gateway, you can build an API to provide your users with an integrated and consistent developer experience to build AWS cloud-based applications.

Part of AWS Serverless Infrastructure

Together with AWS Lambda, API Gateway forms the app-facing part of the AWS serverless infrastructure. For an app to call publicly available AWS services, you can use Lambda to interact with the required services and expose the Lambda functions through API methods in API Gateway. AWS Lambda runs the code on a highly available computing infrastructure. It performs the necessary execution and administration of the computing resources. To enable the serverless applications, API Gateway supports [streamlined proxy integrations \(p. 122\)](#) with AWS Lambda and HTTP endpoints.

Developer Experiences

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

An app developer builds a functioning application to call AWS services by invoking API methods in API Gateway.

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

The app developer does not need to have an AWS account, provided that the API either does not require IAM permissions or supports authorization of users through third-party identity providers supported by [Amazon Cognito 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. Each API includes a set of resources and methods. A resource is a logical entity that an app can access through a resource path.

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.

A method corresponds to a REST API request that is submitted by the user of your API and the response returned to the user. The app does not need to know where the requested data is stored and fetched from on the backend. 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.

As an API developer, you can create and manage an API by using the API Gateway console, described in [Getting Started with Amazon API Gateway \(p. 6\)](#), or by calling the [API Gateway REST API \(p. 581\)](#). There are several ways to call this API. They include using the AWS Command-Line Interface (CLI), or by using an AWS SDK. You can also use a REST API client, such as Postman, to make raw API calls. In addition, you can enable API creation with [AWS CloudFormation templates](#) or [API Gateway Extensions to Swagger \(p. 486\)](#). For a list of regions where API Gateway is available, as well as the associated control service endpoints, see [Regions and Endpoints](#).

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. You can use the API Gateway console to test invoking the API. You can use a REST API client, such as CURL or Postman, or an SDK generated by API Gateway for the API to invoke the API.

Be aware of the differences between the `apigateway` and `execute-api` API Gateway service components. Reference the appropriate service component name when you select one while, for example, setting IAM permission policies for building or calling an API.

Benefits of API Gateway

API Gateway helps you deliver robust, secure, and scalable mobile and web application backends. API Gateway allows you to securely connect mobile and web applications to business logic hosted on AWS Lambda, APIs hosted on Amazon EC2, or other publicly addressable web services hosted inside or outside of AWS. With API Gateway, you can create and operate APIs for backend services. For example, you don't need to develop and maintain infrastructure to handle authorization and access control, traffic management, monitoring and analytics, version management, and software development kit (SDK) generation.

API Gateway is designed for web and mobile developers who want to provide secure, reliable access to backend APIs for access from mobile apps, web apps, and server apps that are built internally or by third-party ecosystem partners. The business logic behind the APIs can be provided by a publicly accessible endpoint that API Gateway proxies call, or it can be entirely run as a Lambda function.

Amazon API Gateway Concepts

API Gateway

API Gateway is an AWS service that supports the following:

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

API Gateway API

A collection of resources and methods that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. The collection can be deployed in one or more stages. API methods are invoked through frontend HTTP endpoints that you can associate with a registered custom domain name. Permissions to invoke a method are granted using IAM roles and policies or

API Gateway Lambda authorizers (formerly known as custom authorizers). An API can present a certificate to be authenticated by the backend. 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 must have unique HTTP verbs supported by API Gateway.

API developer or API owner

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

App developer or client developer

An app creator who may or may not have an AWS account and interacts with the API deployed by the API developer. An app developer can be represented by an API key.

API endpoints

Host names APIs in API Gateway, which are deployed to a specific region and of the `{rest-api-id}.execute-api.{region}.amazonaws.com` format. The following types of API endpoints are supported:

- *Edge-optimized API endpoint*: The default host name 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. An API is edge - optimized if you do not explicitly specify its endpoint type when creating the API.
- *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 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 a region of the lowest latency.

App user, end user, or client

An app user or end user is an entity that uses a client to access an API in Amazon API Gateway. The client can be a mobile app, a web app or a desktop app, using the API Gateway REST API, the AWS CLI or SDK. An app user can be represented by an Amazon Cognito identity or a bearer token.

API key

An alphanumeric string that is generated by API Gateway on behalf of an API owner. The string can also be imported from an external source such as a CSV file. The string is used to identify an app developer of the API. An API owner can use API keys to permit or deny access of specific APIs based on the apps in use.

API deployment and stage

An API deployment is a point-in-time snapshot of the API Gateway API resources and methods. For a deployment to be accessible for a client to invoke, the deployment must be associated with one or more stages. A stage is a logical reference to a lifecycle status of your API (for example, 'dev', 'prod', 'beta', 'v2'). The identifier of an API stage consists of an API ID and stage name.

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 the requests to access the backend through the API.

Integration request

An API Gateway internal interface that defines how API Gateway maps the parameters and body of a method request into the formats required by the backend.

Integration response

An API Gateway internal interface that defines how API Gateway maps data. The integration response includes the status codes, headers, and payload that are received from the backend into the formats defined for an app developer.

Method response

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

Proxy integration

A simplified API Gateway integration configuration. You can set up a proxy integration as an HTTP proxy integration type or a Lambda proxy integration type. For the HTTP proxy integration, API Gateway passes the entire request and response between the frontend and an HTTP backend. For the Lambda proxy integration, API Gateway sends the entire request as an input to a backend Lambda function. API Gateway then transforms the Lambda function output to a frontend HTTP response. The proxy integration is most commonly used with a proxy resource, which is represented by a greedy path variable (e.g., {proxy+}) combined with a catch-all ANY method.

Mapping template

Scripts in [Velocity Template Language \(VTL\)](#) to transform a request body from the frontend data format to the backend data format, or to transform a response body from the backend data format to the frontend data format. Mapping templates are specified in the integration request or integration response. They can reference data made available at run time as context and stage variables. An identity transformation is referred to as a passthrough. In a passthrough, a payload is passed as-is from the client to the backend for a request. For a response, the payload is passed from the backend to the client.

Model

Data schema specifying the data structure of a request or response payload. It is required for generating a strongly typed SDK of an API. It is also used to validate payload. 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.

Usage plan

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

Getting Started with Amazon API Gateway

With Amazon API Gateway, you can provide your clients with a consistent and scalable programming interface to access three types of endpoints in the backend: invoking AWS Lambda functions, calling other AWS services, and accessing an HTTP website or webpage. To do this, you create an API Gateway API to integrate each API method with a backend endpoint. Each backend endpoint is associated with an integration type. For details about the API integration types in API Gateway, see [Choose an API Gateway API Integration Type \(p. 121\)](#).

To get started using Amazon API Gateway, we present the following hands-on walkthroughs for creating, deploying, and testing simple APIs integrated with some commonly used backends. The example APIs used in the walkthroughs demonstrate what is involved to implement each of the supported integration types.

Topics

- [Get Ready to Build an API Gateway API \(p. 6\)](#)
- [Build an API Gateway API from an Example \(p. 9\)](#)
- [Build an API Gateway API with Lambda Integration \(p. 18\)](#)
- [Build an API Gateway API with HTTP Integrations \(p. 38\)](#)
- [Build an API with API Gateway Private Integration \(p. 74\)](#)
- [Build an API Gateway API with AWS Integration \(p. 75\)](#)

Before you start, use the following procedures to set up your development environment.

Get Ready to Build an API Gateway API

Topics

- [Sign up for an AWS Account \(p. 6\)](#)
- [Create IAM Users, Groups, Roles, and Policies in Your AWS Account \(p. 7\)](#)
- [Create IAM Policies to Configure API Gateway Resources and to Call a Deployed API \(p. 7\)](#)
- [Next Step \(p. 9\)](#)

Before using Amazon API Gateway for the first time, you must have an AWS account.

Sign up for an AWS Account

If you do not have an AWS account, use the following procedure to create one.

To sign up for AWS

1. Open <https://aws.amazon.com/> and choose **Create an AWS Account**.

2. Follow the online instructions.

To create, configure, and deploy an API in API Gateway, you must have an appropriate AWS Identity and Access Management policy provisioned. The policy must have access permissions for manipulating the API Gateway [resources](#) and [link relations](#). In addition, you can set IAM permissions to allow your API clients to call your API in API Gateway. To do so, create IAM roles and policies and, optionally, users or groups in your AWS account, and set the IAM roles and policies on a specified IAM user or group.

Create IAM Users, Groups, Roles, and Policies in Your AWS Account

For better security practices, you should create a new AWS Identity and Access Management (IAM) user or use an existing one in your AWS account. You then access API Gateway with that IAM user's credentials, instead of using your AWS root account.

To manage access for a user, create an IAM user and grant the user API Gateway access permissions. To create a new IAM user, see [Creating an IAM User](#).

To manage access for a group of users, create an IAM group, grant the group API Gateway access permissions, and then add one or more IAM users to the group. To create an IAM group, see [Creating IAM Groups](#).

To delegate access to specific users, apps, or services, create an IAM role, add the specified users or groups to the role, and grant the users or groups API Gateway access permissions. To create an IAM role, see [Creating IAM Roles](#).

When setting up your API, specify the ARN of an IAM role to control access the API's methods. This ARN must be ready when creating an API.

Create IAM Policies to Configure API Gateway Resources and to Call a Deployed API

In AWS, access permissions are stated as [IAM policies](#). AWS provides a set of pre-configured IAM policies, known as AWS managed policies, for individual AWS services. Individual IAM users can create customized IAM policies, known as customer managed policies.

You can create an IAM policy, role, user, or group in the IAM console or by using the AWS CLI or an AWS SDK. Once created, IAM policies are referenced by their ARNs. The ARN of a policy that is managed by AWS is of the `arn:aws:iam::aws:policy/PolicyName` format. The ARN of a customer managed policy is of the `arn:aws:iam::123456789012:policy/PolicyName` format.

For example, the following is the AWS managed policy, named **AmazonAPIGatewayAdministrator** (`arn:aws:iam::aws:policy/AmazonAPIGatewayAdministrator`). It grants full access to create, configure, and deploy an API in API Gateway:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "apigateway:*"  
            ],  
            "Resource": "arn:aws:apigateway:*:/*"  
        }  
    ]  
}
```

```
    ]  
}
```

To grant the permissions to a user, attach the policy to the user or a group containing the user. For more information, see [Attaching Managed Policies](#).

Attaching the preceding policy to an IAM user allows ("Effect": "Allow") the user to act with any API Gateway actions ("Action": ["apigateway:*"]) on any API Gateway resources (arn:aws:apigateway::/*) that are associated with the user's AWS account.

To restrict the IAM user to read and create documentation of created APIs, you can replace the Action property value from "Action": ["apigateway:*"] to "Action": ["apigateway:GET", "apigateway:POST"] and replace the Resource property value from ["apigateway:*"] to ["arn:aws:apigateway::123456789012:/restapis/*/documentation/*"]. For more information, see [Control Access to an API with IAM Permissions \(p. 252\)](#).

To control how an API is invoked, the following AWS managed IAM policy of **AmazonAPIGatewayInvokeFullAccess** (arn:aws:iam::aws:policy/AmazonAPIGatewayInvokeFullAccess) provides full access to invoke any part of an API in API Gateway:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "execute-api:Invoke"  
      ],  
      "Resource": "arn:aws:execute-api:*:*:*"  
    }  
  ]  
}
```

To learn how to restrict IAM users to call a specified set of API parts, see [Control Access to an API with IAM Permissions \(p. 252\)](#).

To grant the stated permissions to a user, attach the policy to the user or a group containing the user. To attach a policy, see [Attaching Managed Policies](#).

In this documentation, we use managed policies whenever possible. To create and use customer managed IAM policies, see [Working with Customer Managed Policies](#).

Note

To complete the preceding steps, you must have permissions to create the IAM policy and attach it to the IAM user.

When API Gateway is integrated with AWS Lambda or another AWS service, such as Amazon Simple Storage Service or Amazon Kinesis, you must also enable API Gateway as a trusted entity to invoke an AWS service in the backend. To do so, create an IAM role and attach a service-specific access policy to the role. This is demonstrated in the following example for invoking a Lambda function:

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

```
    ]  
}
```

Next, add the following trust policy to allow API Gateway to call the backend Lambda function on behalf of the attached user who is assigned the IAM role.

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

Without specifying this trust relationship, API Gateway is denied the right to call the backend on behalf of the user, even when the user has been granted permissions to access the backend directly.

When an API Gateway API is set up with IAM roles and policies to control client access, the client must sign API requests with [Signature Version 4](#). Alternatively, you can use the AWS CLI or one of the AWS SDKs to handle request signing for you. For more information, see [Invoking an API in Amazon API Gateway \(p. 457\)](#).

Next Step

Having signed up for an AWS account and created the required IAM roles and policies, you are now ready to start using API Gateway. To create your first simple API by using an example in the API Gateway console, see [Build an API Gateway API from an Example \(p. 9\)](#).

Build an API Gateway API from an Example

To help you get started with basic work flow to build and test an API Gateway API, you can use the Amazon API Gateway console to create and test a simple API with the HTTP integration for a PetStore website. The API definition is preconfigured as a Swagger 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 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>.

- **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 return 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.

Topics

- [Create and Test an API from the Example in the API Gateway Console \(p. 10\)](#)
- [Next Step \(p. 17\)](#)
- [See Also \(p. 18\)](#)

Create and Test an API from the Example in the API Gateway Console

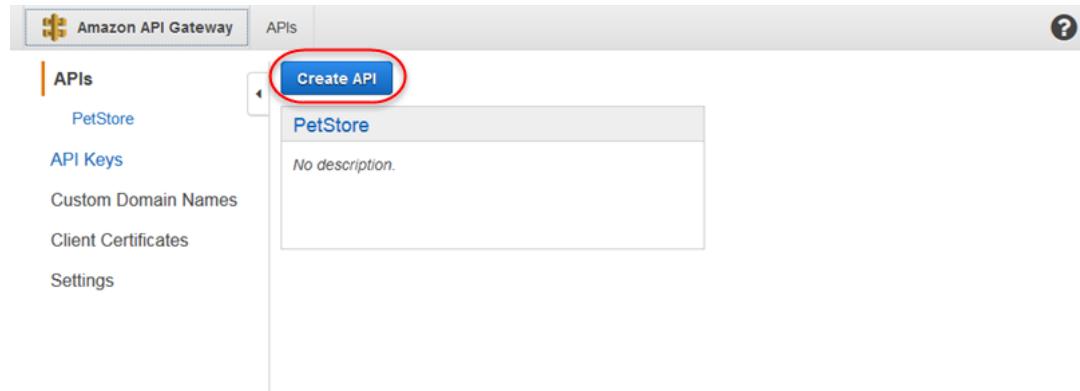
The following procedure walks you through the steps to create and test an API from an example using the API Gateway Console.

To build and test the example API

1. Sign in to the API Gateway console.
2. Do one of the following:
 - a. If this is the first API in your account, choose **Get Started** from the API Gateway console welcome page.

If prompted with hints, choose **OK** to close them and continue.

- b. If this is not your first API, choose **Create API** from the API Gateway APIs home page:



3. Under **Create new API**, choose **Examples API** and then choose **Import** to create the example API. For your first API, the API Gateway console starts with this option as default.

Create new API

In Amazon API Gateway, an API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

New API Clone from existing API Import from Swagger Example API

Example API

Learn about the service by importing an example API and turning on hints throughout the console.

```
1 {
2   "swagger": "2.0",
3   "info": {
4     "description": "Your first API with Amazon API Gateway. This is a sample API that integrates via HTTP with our demo Pet Store endpoints",
5     "title": "PetStore"
6   },
7   "schemes": [
8     "https"
9   ],
10  "paths": {
11    "/": {
12      "get": {
13        "tags": [
14          "pets"
15        ],
16        "description": "PetStore HTML web page containing API usage information",
17        "consumes": [
18          "application/json"
19        ]
20      }
21    }
22  }
23 }
```

Fail on warnings Ignore warnings

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

4. The newly created API is shown as follows:

The screenshot shows the Amazon API Gateway console interface. On the left, the **Resources** pane displays a tree structure of API endpoints:

- / (selected, highlighted in blue)
- /pets
 - GET
 - OPTIONS
 - POST
- /[petId]
 - GET
 - OPTIONS

On the right, the **Actions** dropdown is open, and the **/ Methods** pane is displayed. It shows the details for the selected endpoint:

GET

Mock Endpoint

Authorization: None

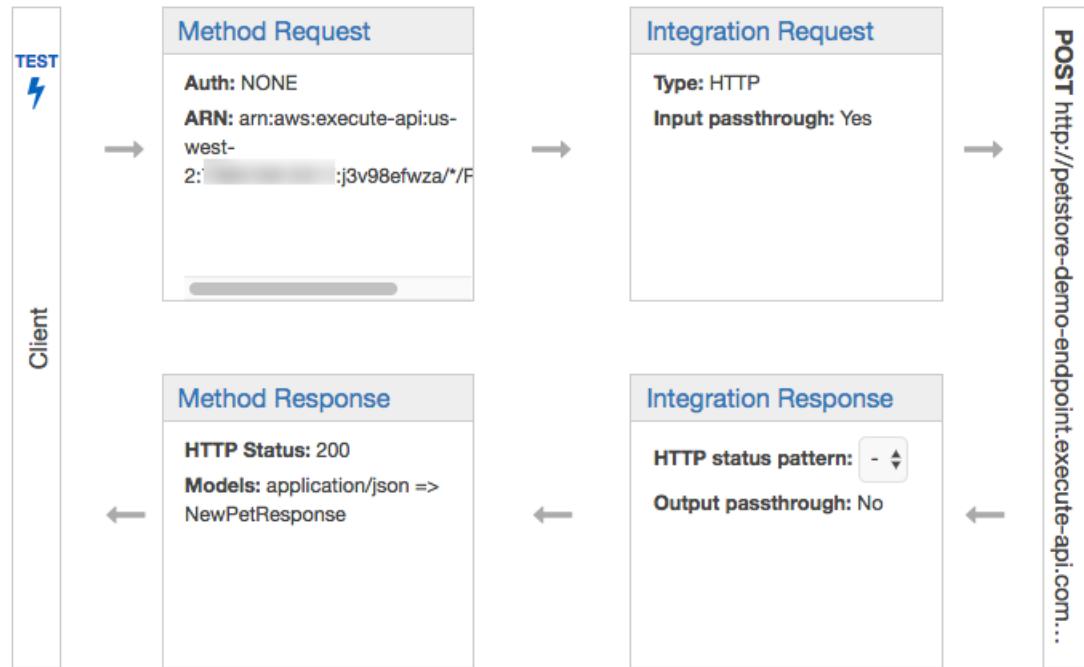
API Key: Not required

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** pane on the right. Displayed under each method is a brief summary of the method, including its endpoint URL, 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:



/pets - POST - Method Execution



The resulting **Method Execution** pane presents a logical view of the chosen (`POST /pets`) method's structure and behaviors: **Method Request** and **Method Response** are the API's interface with the API's frontend (a client), whereas **Integration Request** and **Integration Response** are the API's interface with the backend (`http://petstore-demo-endpoint.execute-api.com/petstore/pets`). 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. We cover the method setup in [Build an API with HTTP Custom Integration \(p. 44\)](#). For now, we focus on testing the API to provide an end-to-end user experience.

6. Choose **Test** shown on **Client** (as shown in the previous image) to start testing. For example, to test the `POST /pets` method, enter the following `{"type": "dog", "price": 249.99}` payload into the **Request Body** before choosing the **Test** button.

[← Method Execution](#)

/pets - POST - Method Test

Make a test call to your method with the provided input

Path

No path parameters exist for this resource. You can define path parameters by using the syntax {myPathParam} in a resource path.

Query Strings

No query string parameters exist for this method. You can add them via Method Request.

Headers

No header parameters exist for this method. You can add them via Method Request.

Stage Variables

No [stage variables](#) exist for this method.

Client Certificate

No client certificates have been generated.

Request Body

```
1  [{"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.

7. The results display as follows:

Request: /pets

Status: 200

Latency: 566 ms

Response Body

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

Response Headers

```
{"Access-Control-Allow-Origin":"*","X-Amzn-Trace-Id":"Root=1-59287e14-bd5f1d07c673367be0739eae","Content-Type":"application/json"}
```

Logs

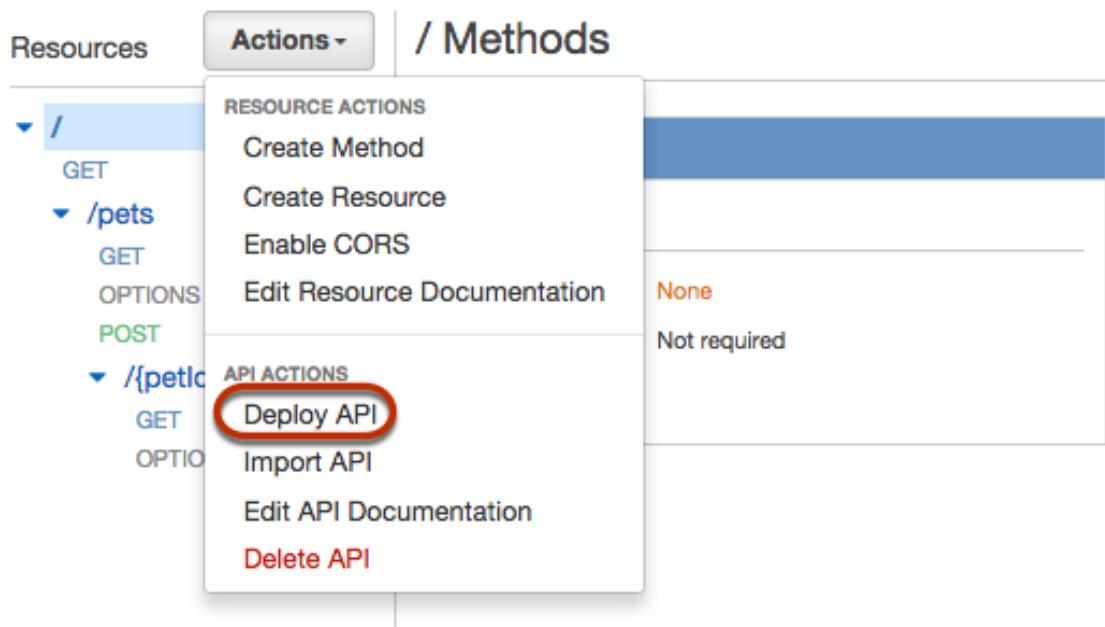
```
Execution log for request test-request
Fri May 26 19:12:20 UTC 2017 : Starting execution for request: test-invoke-request
Fri May 26 19:12:20 UTC 2017 : HTTP Method: POST, Resource Path: /pets
Fri May 26 19:12:20 UTC 2017 : Method request path: {}
Fri May 26 19:12:20 UTC 2017 : Method request query string: {}
Fri May 26 19:12:20 UTC 2017 : Method request headers: {}
Fri May 26 19:12:20 UTC 2017 : Method request body before transformations: {"type": "dog", "price": 249.99}
Fri May 26 19:12:20 UTC 2017 : Endpoint request URI: http://petstore-demo-endpoint.execute-api.com/petstore/pets
Fri May 26 19:12:20 UTC 2017 : Endpoint request headers: {x-amzn-apigateway-api-id=4wk1k4onj3, Accept=application/json, User-Agent=AmazonAPIGateway_4wk1k4onj3, X-Amzn-Trace-Id=Root=1-59287e14-bd5f1d07c673367be0739eae, Content-Type=application/json}
Fri May 26 19:12:20 UTC 2017 : Endpoint request body after transformations: {"type": "dog", "price": 249.99}
Fri May 26 19:12:20 UTC 2017 : Endpoint response body before transformations: {
  "pet": {
    "type": "dog",
    "price": 249.99
  },
  "message": "success"
}
Fri May 26 19:12:20 UTC 2017 : Endpoint response headers: {Connection=keep-alive, Content-Length=81, Date=Fri, 26 May 2017 19:12:20 GMT, Content-Type=application/json; charset=utf-8, X-Powered-By=Express}
Fri May 26 19:12:20 UTC 2017 : Method response body after transformations: {
  "pet": {
    "type": "dog",
    "price": 249.99
  },
  "message": "success"
}
Fri May 26 19:12:20 UTC 2017 : Method response headers: {Access-Control-Allow-Origin=*, X-Amzn-Trace-Id=Root=1-59287e14-bd5f1d07c673367be0739eae, Content-Type=application/json}
Fri May 26 19:12:20 UTC 2017 : Successfully completed execution
Fri May 26 19:12:20 UTC 2017 : Method completed with status: 200
```

The **Logs** 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.

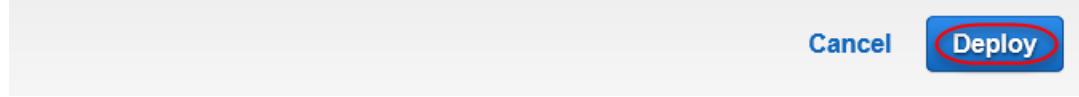
8. To deploy the sample API, select the **PetStore** API, and then choose **Deploy API** from the **Actions** menu.



In **Deploy API**, for **Deployment stage**, choose **[New Stage]** because this is the first deployment of the API. Type a name (e.g., `test`) in **Stage name** and, optionally, type descriptions in **Stage description** and **Deployment description**. Choose **Deploy**.



Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.



In the resulting **Stage Editor** pane, **Invoke URL** displays the URL to invoke the API's `GET /` method request.

9. On **Stage Editor**, follow the **Invoke URL** link to submit the `GET /` method request in a browser. A successful response return the result, generated from the mapping template in the integration response.
10. 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 protocols. For an example of such a request, see [Build an API with HTTP Custom Integration \(p. 44\)](#).

Next Step

Through the example API, we became familiar with the basic workflow for creating an API in API Gateway. The process is summarized as follows:

1. Create an API as a [RestApi](#) resource in your AWS account.
2. Add a [Resource](#) resource to the Resources hierarchy of the newly created API.
3. Create a [Method](#) resource for the [Resource](#). The API method represents a programming interface between a client and API Gateway.
4. Set up the integration of the method with a backend endpoint. The integration represents an interface between the API Gateway and a backend endpoint.

When a user accesses the backend service through the API, the client submits an HTTP request to API Gateway. This submission puts the request through the [Method Request](#) and then [Integration Request](#) before reaching the backend. The backend then returns a response to API Gateway. The response then passes from [Integration Response](#) to [Method Response](#) before the client receives the response. The [MOCK](#) integrations demonstrated in this example API are perhaps the simplest cases of pre-processing and post-processing of requests or responses by API Gateway. We cover other cases elsewhere in this guide.

Next, we move on to learning how to build and test a more nimble and powerful API with [proxy integrations \(p. 126\)](#).

See Also

[Use API Gateway Lambda Authorizers \(p. 272\)](#), [Deploying an API in Amazon API Gateway \(p. 370\)](#)

Build an API Gateway API with Lambda Integration

To build an API with Lambda integrations, you can use either the Lambda proxy integration or the Lambda custom integration. In general, you should use the Lambda proxy integration for a nimble and streamlined API set up while providing versatile and powerful features. The custom integration may be a better value proposition if it is necessary for API Gateway to pre-process incoming request data before it reaches to the backend Lambda function. However, it is a legacy technology. Setting up a Lambda custom integration is more involved than setting up the Lambda proxy integration and the existing setup is likely to be inoperable when the backend Lambda function requires changes in its input or output.

With the 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 the 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.

With the Lambda custom integration, you must ensure that the 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 may also need to translate the client-supplied request body into a format recognized by the Lambda function.

Topics

- [Build an API Gateway API with Lambda Proxy Integration \(p. 19\)](#)
- [Build an API Gateway API with Cross-Account Lambda Proxy Integration \(p. 27\)](#)
- [Build an API Gateway API with Custom Lambda Integration \(p. 29\)](#)

Build an API Gateway API with Lambda Proxy Integration

In this section, we show how to create and test an API with Lambda integration using the API Gateway console. We demonstrate how a Lambda backend parses the raw request and implements app logic that depends on the incoming request data. For more information on API Gateway proxy integration, see [Set up a Proxy Integration with a Proxy Resource \(p. 122\)](#).

First, we create the following Node.js function, named `GetStartedLambdaProxyIntegration`, using the AWS Lambda console, as the backend. We then create an API with the Lambda proxy integration by using the `GetStartedLambdaProxyIntegration` function through a proxy resource by using the API Gateway console. Finally, we demonstrate how to test the API.

Topics

- [Create Lambda Functions for an API with Lambda Proxy Integration \(p. 19\)](#)
- [Create a Backend for an API with Lambda Proxy Integration \(p. 23\)](#)
- [Create an API with Lambda Proxy Integration \(p. 24\)](#)
- [Test an API with Lambda Proxy Integration \(p. 25\)](#)

Create Lambda Functions for an API with Lambda Proxy Integration

We create a Lambda function that returns a greeting to the caller as a JSON object of the following format:

```
{  
    "greeting": "Good {time}, {name} of {city}.[ Happy {day}]"  
}
```

In this example, `{time}` can be `morning`, `afternoon`, or `day`; `{name}` can be `you` or a user-specified user name; `{city}` can be `World` or a user-supplied city name; and `{day}` can be `null`, empty, or one of the week days. If `{day}` is `null` or empty, the `Happy {day}` portion is not displayed. The Lambda function is very flexible and the client can specify the input in any combination of request headers, path variables, query string parameters, and body.

To show what API Gateway passes through to the backend, we include the `event` object to the Lambda function in its output as well. Finally, we create a `response` object to illustrate the basic output format required of the Lambda proxy integration.

A Lambda function can be written in Node.js, Python, Java, and C#. In this tutorial, we show snippets in Node.js and Java. You can extend the Node.js implementation to the Python function or extend the Java implementation to the C# function. There are instructions for doing so in the following topics.

Topics

- [Node.js Function for an API with Lambda Proxy Integration \(p. 20\)](#)
- [Python Function for an API with Lambda Proxy Integration \(p. 21\)](#)
- [C# Function for an API with Lambda Proxy Integration \(p. 21\)](#)
- [Java Function for an API with Lambda Proxy Integration \(p. 21\)](#)

Node.js Function for an API with Lambda Proxy Integration

The following Lambda function in Node.js is a "Hello, World!" application. The function shows how to parse the input event parameter that contains a request made by a client to an API Gateway proxy resource. This resource is integrated with the function using the Lambda proxy integration. The function also demonstrates how to format the output of the Lambda function for API Gateway to return the results as an HTTP response. For more information about the input and output formats that this type of Lambda function must follow, see [Input Format of a Lambda Function for Proxy Integration \(p. 132\)](#) and [Output Format of a Lambda Function for Proxy Integration \(p. 134\)](#).

```
'use strict';
console.log('Loading hello world function');

exports.handler = function(event, context, callback) {
    let name = "you";
    let city = 'World';
    let time = 'day';
    let day = '';
    let responseCode = 200;
    console.log("request: " + JSON.stringify(event));

    // This is a simple illustration of app-specific logic to return the response.
    // Although only 'event.queryStringParameters' are used here, other request data,
    // such as 'event.headers', 'event.pathParameters', 'event.body',
    'event.stageVariables',
    // and 'event.requestContext' can be used to determine what response to return.
    //
    if (event.queryStringParameters !== null && event.queryStringParameters !== undefined) {
        if (event.queryStringParameters.name !== undefined &&
            event.queryStringParameters.name !== null &&
            event.queryStringParameters.name !== "") {
            console.log("Received name: " + event.queryStringParameters.name);
            name = event.queryStringParameters.name;
        }
    }

    if (event.pathParameters !== null && event.pathParameters !== undefined) {
        if (event.pathParameters.proxy !== undefined &&
            event.pathParameters.proxy !== null &&
            event.pathParameters.proxy !== "") {
            console.log("Received proxy: " + event.pathParameters.proxy);
            city = event.pathParameters.proxy;
        }
    }

    if (event.headers !== null && event.headers !== undefined) {
        if (event.headers['day'] !== undefined && event.headers['day'] !== null &&
            event.headers['day'] !== "") {
            console.log("Received day: " + event.headers.day);
            day = event.headers.day;
        }
    }

    if (event.body !== null && event.body !== undefined) {
        let body = JSON.parse(event.body)
        if (body.time)
            time = body.time;
    }

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

    var responseBody = {
        statusCode: responseCode,
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(greeting)
    };
    callback(null, responseBody);
}
```

```

        message: greeting,
        input: event
    };

    // The output from a Lambda proxy integration must be
    // of the following JSON object. The 'headers' property
    // is for custom response headers in addition to standard
    // ones. The 'body' property must be a JSON string. For
    // base64-encoded payload, you must also set the 'isBase64Encoded'
    // property to 'true'.
    var response = {
        statusCode: responseCode,
        headers: {
            "x-custom-header" : "my custom header value"
        },
        body: JSON.stringify(responseBody)
    };
    console.log("response: " + JSON.stringify(response))
    callback(null, response);
}

```

For the API Gateway proxy integrations, the input parameter of event contains an API request marshalled as a JSON object by API Gateway. This input can include the request's HTTP method (`httpMethod`), path (`path` and `pathParameters`), query parameters (`queryStringParameters`), headers (`headers`), and applicable payload (`body`). The input can also include the context (`requestContext`) and stage variables (`stageVariables`).

This example Lambda function parses the event parameter to retrieve the query string parameter of `name`, the proxy path parameter, the `day` header value, and the `time` property of the payload.

The function then returns a greeting to the named user in the `message` property of the `responseBody` object. To show the details of the incoming request as marshalled by API Gateway, the function also returns the incoming event object as the `input` property of the response body.

Finally, upon exiting, the function returns a JSON object, containing the required `statusCode` and any applicable `headers` and `body`, for API Gateway to return it as an HTTP response to the client.

Python Function for an API with Lambda Proxy Integration

Follow the discussion in [Authoring Lambda Functions in Python](#) to create the Python Lambda function handler, while extending the programming flow shown in the preceding Node.js Lambda function.

C# Function for an API with Lambda Proxy Integration

Follow the discussion in [Authoring Lambda Functions in C#](#) to create the C# Lambda function handler, while extending the programming flow shown in the following Java Lambda function.

Java Function for an API with Lambda Proxy Integration

The following Lambda function in Java is a "Hello, World!" application, similar to its Node.js counterpart ([p. 20](#)). The function shows how to parse the input event that is passed through as an `InputStream` object and that contains a request made by a client to an API Gateway proxy resource. This resource is integrated with the function using the Lambda proxy integration. It also shows how to parse the `context` object to get the `LambdaLogger`. The example also demonstrates how to format the output of the Lambda function for API Gateway in Java to return the results in an `OutputStream` object as an HTTP response. For more information about the Lambda proxy integration input and output formats, see [Input Format of a Lambda Function for Proxy Integration \(p. 132\)](#) and [Output Format of a Lambda Function for Proxy Integration \(p. 134\)](#).

```
package examples;
```

```

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.BufferedReader;
import java.io.Writer;

import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

import org.json.simple.JSONObject;
import org.json.simple.JSONArray;
import org.json.simple.parser.ParseException;
import org.json.simple.parser.JSONParser;
}

public class ProxyWithStream implements RequestStreamHandler {
    JSONParser parser = new JSONParser();

    public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws IOException {
        LambdaLogger logger = context.getLogger();
        logger.log("Loading Java Lambda handler of ProxyWithStream");

        BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
        JSONObject responseJson = new JSONObject();
        String name = "you";
        String city = "World";
        String time = "day";
        String day = null;
        String responseCode = "200";

        try {
            JSONObject event = (JSONObject)parser.parse(reader);
            if (event.get("queryStringParameters") != null) {
                JSONObject qps = (JSONObject)event.get("queryStringParameters");
                if (qps.get("name") != null) {
                    name = (String)qps.get("name");
                }
            }

            if (event.get("pathParameters") != null) {
                JSONObject pps = (JSONObject)event.get("pathParameters");
                if (pps.get("proxy") != null) {
                    city = (String)pps.get("proxy");
                }
            }

            if (event.get("headers") != null) {
                JSONObject hps = (JSONObject)event.get("headers");
                if (hps.get("day") != null) {
                    day = (String)hps.get("day");
                }
            }

            if (event.get("body") != null) {
                JSONObject body = (JSONObject)parser.parse((String)event.get("body"));
                if (body.get("time") != null) {
                    time = (String)body.get("time");
                }
            }
        }
    }
}

```

```

        }

        String greeting = "Good " + time + ", " + name + " of " + city + ". ";
        if (day!=null && day != "") greeting += "Happy " + day + "!";

        JSONObject responseBody = new JSONObject();
        responseBody.put("input", event.toJSONString());
        responseBody.put("message", greeting);

        JSONObject headerJson = new JSONObject();
        headerJson.put("x-custom-header", "my custom header value");

        responseJson.put("isBase64Encoded", false);
        responseJson.put("statusCode", responseCode);
        responseJson.put("headers", headerJson);
        responseJson.put("body", responseBody.toString());

    } catch(ParseException pex) {
        responseJson.put("statusCode", "400");
        responseJson.put("exception", pex);
    }

    logger.log(responseJson.toJSONString());
    OutputStreamWriter writer = new OutputStreamWriter(outputStream, "UTF-8");
    writer.write(responseJson.toJSONString());
    writer.close();
}
}

```

For proxy integrations in API Gateway, the input stream contains an API request serialized as a JSON string by API Gateway. The input data can include the request's HTTP method (`httpMethod`), path (`path` and `pathParameters`), query parameters (`queryStringParameters`), headers (`headers`), applicable payload (`body`), the context (`requestContext`), and stage variables (`stageVariables`).

This example Lambda function parses the `inputStream` parameter to retrieve the `query string` parameter of `name`, the `proxy` path parameter, the `day` header value and the `time` property of the payload. For logging, it retrieves the `LambdaLogger` object from the incoming `context` object.

The function then returns a greeting to the named user in the `message` property of the `responseBody` object. To show the details of the incoming request as marshalled by API Gateway, the function also returns the input data (`event`) in the response body.

Finally, upon exiting, the function returns a JSON string, containing the required `statusCode` and any applicable `headers` and `body`, for API Gateway to return it as an HTTP response to the client.

To create this function in the Lambda console, you must create a deployment package before uploading the package into Lambda. For more information, see [creating a deployment package in the AWS Lambda Developer Guide](#).

Create a Backend for an API with Lambda Proxy Integration

The following procedure describes how to create the Lambda function in API Gateway using the Lambda console.

Create a Lambda function for an API with a proxy resource in the Lambda console

1. Sign in to the Lambda console at <https://console.aws.amazon.com/lambda>.
2. From the upper-right corner, choose an available region for the Lambda function.
3. From the main navigation pane, choose **Functions**. You may need to choose the navigation menu on the top-left corner if the navigation pane is not displayed.

4. Choose **Create function**. And then choose **Author from scratch** or **Blueprints**. For this example, we create a function from scratch.
5. Under **Author from scratch**, do the following:
 - a. In the **Name** input field, type a function name.
 - b. From the **Runtime** drop-down list, choose a supported runtime. In this example, we use **Node.js 4.3**.
 - c. From the **Role** drop-down list, choose **Choose an existing role**, **Create new role from template(s)** or **Create a custom role**. Then, follow the ensuing instructions for the choice.
 - d. Choose **Create function** to continue.

For this example, we will skip the **Designer** section and move to the **Function code** section next.

6. For a Node or Python runtime, you can use the inline code editor to create or edit the lambda function, in addition to uploading a zipped code file from a local drive or from Amazon S3. For a Java or C# runtime, you must upload the zipped code file from a local drive or from Amazon S3. In any case, use the code example of the specified runtime as specified in [the section called "Create Lambda Functions for an API with Lambda Proxy Integration" \(p. 19\)](#) here.
7. Choose **Save** to finish creating the Lambda function.
8. Optionally, but highly recommended, choose **Test** and configure the test event to take the required [Lambda proxy integration request input \(p. 132\)](#).

Note

Note the region where you created the Lambda function. You need it when creating the API for the function.

Create an API with Lambda Proxy Integration

Now create an API with a proxy resource for a Lambda function by using the API Gateway console.

Build an API with a proxy resource for a Lambda function

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. To create an API, choose **Create new API** (for creating the first API) or **Create API** (for creating any subsequent API). Next, do the following:
 - a. Choose **New API**.
 - b. Type a name in **API Name**.
 - c. Optionally, add a brief description in **Description**.
 - d. Choose **Create API**.

For this tutorial, use `LambdaSimpleProxy` as the API name.

3. To create a child resource, choose a parent resource item under the **Resources** tree and then choose **Create Resource** from the **Actions** drop-down menu. Then, do the following in the **New Child Resource** pane.
 - a. Select the **Configure as proxy resource** option to create a proxy resource. Otherwise, leave it de-selected.
 - b. Type a name in the **Resource Name*** input text field.
 - c. Type a new name or use the default name in the **Resource Path*** input text field.
 - d. Choose **Create Resource**.
 - e. Select **Enable API Gateway CORS**, if required.

For this tutorial, use the root resource (/) as the parent resource. Select **Configure as proxy resource**. For **Resource Name**, use the default, proxy. For **Resource Path**, use /{proxy+}. De-select **Enable API Gateway CORS**.

4. To set up the ANY method for integration with the Lambda back end, do the following:
 - a. Choose the resource just created and then choose **Create Method** from the **Actions** drop-down menu.
 - b. Choose **ANY** from the HTTP method drop-down list and then choose the check mark icon to save the choice.
 - c. Choose **Lambda Function Proxy** for **Integration type**.
 - d. Choose a region from **Lambda Region**.
 - e. Type the name of your Lambda function in **Lambda Function**.
 - f. Choose **Save**.
 - g. Choose **OK** when prompted with **Add Permission to Lambda Function**.

For this tutorial, use the previously created [GetStartedLambdaProxyIntegration \(p. 20\)](#) for the **Lambda Function**.

For the proxy resource API that Lambda just created, API Gateway forwards the raw request from the client to the backend for the Lambda function to process. The request includes the request method, its path, query string and headers parameters, any payload, plus context and stage variables. The next procedure describes how to test this.

Test an API with Lambda Proxy Integration

The following procedure describes how to test the proxy integration.

Call the [GetStartedLambdaProxyIntegration \(p. 20\)](#) Lambda function through the proxy resource

- To use a browser to call a GET method on a specific resource of the API, do the following.
 - a. If you have not done so, choose **Deploy API** from the **Actions** drop-down menu for the API you created. Follow the instructions to deploy the API to a specific stage. Note the **Invoke URL** that displays on the resulting **Stage Editor** page. This is the base URL of the API.
 - b. To submit a GET request on a specific resource, append the resource path, including possible query string expressions to the **Invoke URL** value obtained in the previous step, copy the complete URL into the address bar of a browser, and choose Enter.

For this tutorial, deploy the API to a **test** stage and note of the API's base URL; for example, <https://wt6mne2s9k.execute-api.us-west-2.amazonaws.com/test>.

There are several ways you can test a deployed API. For GET requests using only URL path variables or a query string parameter, you can type 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 deployed 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://r275xc9bmd.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:

```
{
  "message": "Good day, John of Seattle. Happy Friday!",
  "input": {
    "resource": "/{proxy+}",
    "path": "/Seattle",
    "httpMethod": "POST",
    "headers": {
      "day": "Friday"
    },
    "queryStringParameters": {
      "time": "morning"
    },
    "pathParameters": {
      "proxy": "Seattle"
    },
    "stageVariables": null,
    "requestContext": {
      "path": "/{proxy+}",
      "accountId": "123456789012",
      "resourceId": "n19h80",
      "stage": "test-invoke-stage",
      "requestId": "test-invoke-request",
      "identity": {
        "cognitoIdentityPoolId": null,
        "accountId": "123456789012",
        "cognitoIdentityId": null,
        "caller": "AIDXXX...XXVJZG",
        "apiKey": "test-invoke-api-key",
        "sourceIp": "test-invoke-source-ip",
        "accessKey": "ASIXXX...XXDQ5A",
        "cognitoAuthenticationType": null,
        "cognitoAuthenticationProvider": null,
        "userArn": "arn:aws:iam::123456789012:user/kdeding",
        "userAgent": "Apache-HttpClient/4.5.x (Java/1.8.0_131)",
        "user": "AIDXXX...XXVJZG"
      },
      "resourcePath": "/{proxy+}",
      "httpMethod": "POST",
      "apiId": "r275xc9bmd"
    },
    "body": "{ \"callerName\": \"John\" }",
    "isBase64Encoded": false
  }
}
```

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

3. To test **GET /Boston?time=morning**:

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

```
curl -X GET \
  'https://r275xc9bmd.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'
```

You get a 200 OK Request response similar to the result from the preceding POST request, with the exception that the GET request does not have any payload. So the body parameter will be null.

Note

The `requestContext` is a map of key-value pairs and corresponds to the [\\$context \(p. 191\)](#) variable. Its outcome is API-dependent. API Gateway may add new keys to the map. For more information, see [Input Format of a Lambda Function for Proxy Integration \(p. 132\)](#).

Build an API Gateway 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.

First, we create the example API from [the section called “Build an API from an Example” \(p. 9\)](#) in one account. We then create a Lambda function in another account. Finally, we use cross-account Lambda integration to allow the example API to use the Lambda function we created in the second account.

Create API for API Gateway Cross-Account Lambda Integration

First, you'll create the example API as described in [the section called “Build an API from an Example” \(p. 9\)](#).

To create the example API

1. Sign in to the API Gateway console.
2. Choose **Create API** from the API Gateway APIs home page:
3. Under **Create new API**, choose **Examples API**.
4. For **Endpoint Type**, choose **Edge optimized**.
5. Choose **Import** to create the example 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. In the **Name** input field, type a function name.
 - b. From the **Runtime** drop-down list, choose a supported runtime. In this example, we use **Node.js 6.10**.
 - c. From the **Role** drop-down list, choose **Choose an existing role**, **Create new role from template(s)** or **Create a custom role**. Then, follow the ensuing instructions for the choice.
 - d. Choose **Create function** to continue.

For this example, we will skip the **Designer** section and move to the **Function code** section next.

5. Scroll down to the **Function code** pane.
6. Copy-paste a function implementation such as one of the API Gateway examples for [Node.js \(p. 20\)](#) and [Java \(p. 21\)](#).
7. Choose the correct runtime from the **Runtime** drop-down menu.
8. Choose **Save**.
9. 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 the API Gateway console to add it to your API in your first account.

Configuring your cross-account Lambda integration

1. In the API Gateway console, choose your API.
2. Choose **Resources**.
3. In the **Resources** pane, choose the top-level **GET** method.
4. In the **Method Execution** pane, choose **Integration Request**.
5. For **Integration type**, choose **Lambda Function**.
6. Check **Use Lambda Proxy integration**.
7. Leave **Lambda Region** set to your account's region.
8. For **Lambda Function**, copy/paste the full ARN for the Lambda function you created in your second account and choose the checkmark.
9. You'll see a popup that says **Add Permission to Lambda Function: You have selected a Lambda function from another account. Please ensure that you have the appropriate Function Policy on this function. You can do this by running the following AWS CLI command from account 123456789012**; followed by an `aws lambda add-permission` command string.
10. Copy-paste the `aws lambda add-permission` command string into an AWS CLI window that is configured for your second account. This will grant your first account access to your second account's Lambda function.
11. In the popup from the previous step in the Lambda console, choose **OK**.
12. To see the updated policy for your function in the Lambda console,

- a. Choose your integration function.
- b. In the **Designer** pane, choose the key icon.

In the **Function policy** pane, you should now see an **Allow** policy with a **Condition** clause in which the **in the AWS:SourceArn** is the ARN for your API's GET method.

Build an API Gateway API with Custom Lambda Integration

Note

The *Lambda custom integration*, formerly known as the *Lambda integration*, is a legacy technology. We recommend that you use the *Lambda proxy integration* for any new API. For more information, see the section called "Build an API with Lambda Proxy Integration" (p. 19).

In this walkthrough, we use the API Gateway console to build an API that enables a client to call Lambda functions through the Lambda 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 [Create an API Gateway API for AWS Lambda Functions \(p. 507\)](#).

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

Topics

- [Create a Lambda Function for the Lambda Custom Integration \(p. 29\)](#)
- [Create an API with the Lambda Custom Integration \(p. 33\)](#)
- [Test Invoking the API Method \(p. 35\)](#)
- [Deploy the API \(p. 36\)](#)
- [Test the API in a Deployment Stage \(p. 37\)](#)
- [Clean Up \(p. 38\)](#)

Create a Lambda Function for the Lambda Custom 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`. It is similar to [GetStartedLambdaProxyIntegration \(p. 20\)](#), which is the function we created for the Lambda proxy integration.

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

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

```

console.log('Loading function');

exports.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
    });
};

```

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](#).

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 Swagger](#), 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 \(p. 252\)](#).

Compared to [GetStartedLambdaProxyIntegration \(p. 20\)](#), 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 a function**.
 - If the **Lambda > Functions** list page appears, choose **Create a function**.
3. From **Select blueprint**, choose **Author from scratch**.
4. In the **Configure triggers** pane, choose **Next**.
5. In the **Configure function** pane, do the following:
 - a. Under **Basic information**:
 - For **Name**, type `GetStartedLambdaIntegration` as the Lambda function name.
 - For **Description**, type `Backend for the Getting Started walkthrough with Lambda custom integration`. This is optional and you can leave it blank.
 - For **Runtime**, choose `Node.js 6.10`.
 - b. Under **Lambda function code**:
 - Choose **Edit code inline**, if it is not already shown, under **Content entry type**.
 - Copy the Lambda function code listed in the beginning of this section and paste it in the inline code editor.
 - Leave the default choices for all other fields in this section.
 - c. Under **Lambda function handler and role**:
 - Leave the default of `index.handler` for **Handler**.
 - For **Role**, choose **Create new role from template(s)**.
 - For **Role name**, type a name for your role (for example, `GetStartedLambdaIntegrationRole`).
 - For **Policy templates**, choose **Simple Microservice permissions**.
6. To test the newly created function, as a best practice, choose **Actions** and select **Configure test event**.
 - a. For **Input test event**, replace any default code statements with the following, and then choose **Save and test**.

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

```
}
```

- b. Choose **Test** to invoke the function. The **Execution result: succeeded** section is shown. Expand **Detail** 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 the Lambda Custom 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.

The API is named `GetStartedLambdaIntegrationAPI`.

To create an API with Lambda custom integration with a Lambda function

1. Launch the API Gateway console.
2. Choose **Create new API**.
 - a. Type `GetStartedLambdaIntegrationAPI` for **API name**.
 - b. Type a description of the API for **Description** or leave it blank.
 - c. Choose **Create API**.
3. Choose the root resource `(/)` under **Resources**. From the **Actions** menu, choose **Create Resource**.
 - a. Type `city` for **Resource Name**.
 - b. Replace **Resource Path** with `{city}`. This is an example of the templated path variable used to take input from the client. Later, we show how to map this path variable into the Lambda function input using a mapping template.
 - c. Select the **Enable API Gateway Cors** option.
 - d. Choose **Create Resource**.
4. With the newly created `{city}` resource highlighted, choose **Create Method** from **Actions**.
 - a. Choose **ANY** from the HTTP method drop-down menu. 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.
 - b. To save the setting, choose the check mark.
5. In **Method Execution**, for the `ANY /{city}` method, do the following:
 - a. Choose **Lambda Function** for **Integration type**.
 - b. Leave the **Use Lambda Proxy integration** box clear to use custom Lambda custom integration.
 - c. Choose the region where you created the Lambda function; for example, `us-west-2`.
 - d. Type the name of your Lambda function in **Lambda Function**; for example, `GetStartedLambdaIntegration`.
 - e. Choose **Save**.
 - f. Choose **OK** in **Add Permission to Lambda Function** to have API Gateway set up the required access permissions for the API to invoke the integrated Lambda function.
6. In **Method Execution**, choose **Method Request** and configure as follows:

- A query string parameter (`time`)
- To set up a header parameter (`day`)
- To define a payload property (`callerName`)

At run time, the client can use these request parameters and the request body to provide time of the day, the day of the week, and the name of the caller. You already configured the `/{city}` path variable.

- a. Under **Settings** choose the pencil icon to choose Validate body, query string parameters, and headers from the **Request Validator** drop-down menu. This lets API Gateway perform basic request validation before forwarding the request to the Lambda function.
- b. Expand the **URL Query String Parameters** section. Choose **Add query string**. Type `time` for **Name**. Select the **Required** option and choose the check-mark icon to save the setting. Leave **Caching** cleared to avoid an unnecessary charge for this exercise.
- c. Expand the **HTTP Request Headers** section. Choose **Add header**. Type `day` for **Name**. Select the **Required** option and choose the check-mark icon to save the setting. Leave **Caching** cleared to avoid an unnecessary charge for this exercise.
- d. To define the method request payload, do the following:
 - i. To define a model, choose **Models** under the API from the API Gateway primary navigation pane, and then choose **Create**.
 - ii. Type `GetStartedLambdaIntegrationUserInput` for **Model name**.
 - iii. Type `application/json` for **Content type**.
 - iv. Type a description for **Model description** or leave it blank.
 - v. Copy the following schema definition to the **Model schema** editor:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "GetStartedLambdaIntegrationInputModel",
  "type": "object",
  "properties": {
    "callerName": { "type": "string" }
  }
}
```

- vi. Choose **Save** to finish defining the input model.
- vii. Go back to **Method Request** and expand **Request body**. Choose **Add model**. Type `application/json` for **Content type**. Choose `GetStartedLambdaIntegrationInput` for **Model name**. Choose the check-mark icon to save the setting.
7. In **Method Execution** for the ANY `/{city}` method, choose **Integration Request** to set up a body-mapping template. This maps the previously configured method request parameter of `nameQuery` or `nameHeader` to the JSON payload, as required by the backend Lambda function.
 - a. Expand the **Body Mapping Templates** section. Choose **Add mapping template**. Type `application/json` for **Content-Type**. Choose the check-mark icon to save the setting.
 - b. Type the following mapping template in the VTL script editor. Choose **Save** to finish the setup.
8. Choose **Integration Request** to set up a mapping template to transform the client-supplied request data to the input format of the integrated Lambda function:
 - a. Expand the **Body mapping templates** section.
 - b. Check the recommended when there are no templates defined for **Request body passthrough**.

- c. Choose **Add mapping template**.
- d. Type application/json for **Content-type**.
- e. Choose the check-mark icon to save the setting.
- f. Choose GetStartedLambdaIntegrationUserInput from **Generate template** to generate an initial mapping template. This option is available because you defined a model schema, without which you would need to write the mapping template from scratch.
- g. Modify the mapping script in the mapping template editor as follows:

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

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 set ANY to POST, set {city} to Seattle, assign Wednesday as the day header value, and assign "John" as the callerName value.

To test invoking the ANY /{city} method

1. In **Method Execution**, choose **Test**.
2. Choose **POST** from the **Method** drop-down list.
3. Type **Seattle** for the **{city}** path variable.
4. Type **morning** for the **day** query string parameter.
5. Type **{ "callerName": "John" }** for **Request Body**.
6. Choose **Test**.
7. Verify that the returned response payload is as follows:

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

8. 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=FOoDYXdzELL//////////wEaDMHGzEdEOT/VvGhabiK3AzgKrJw
  +3zLqJZG4PhOq12K6W21+QotY2rrZyOzqhLoiuRg3CAYNQ2eqgL5D54+63ey9bIdtwHGo
  yBdq8ecWxJK/YUnT2Rau0L9HCG5p7FC05h3Ivw1FfvcidQNxeYvsKJTLXI05/
  yEnY3ttiAwpNYLOezD9Es8rBfyruHfJf0qextKlsC8DymCcqlGkig8qLKCZ0hWJWWwiP
  PJiFgL7laabXs++ZhCa4hdZo4iqlG729DE4gaV1mJVdoAagIUwLMo+yNxFDu0r7I0/
  EO5nYcCrppGVVBYiGk7H4T6sXuhTkbNNqVmXtV3ch5b0lh7 [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 the API from the **APIs** pane or choose a resource or method from the **Resources** pane. Choose **Deploy API** from the **Actions** drop-down menu.
2. For **Deployment stage**, choose **New Stage**.

3. For **Stage name**, type a name; for example, **test**.

Note

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

4. For **Stage description**, type a description or leave it blank.
5. For **Deployment description**, type a description or leave it blank.
6. Choose **Deploy**. After the API is successfully deployed, you see the API's base URL (the default host name plus the stage name) displayed as **Invoke URL** at the top of the **Stage Editor**. The general pattern of this 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 type 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.

3. To test GET /Boston?time=morning:

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

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

You get a 400 Bad Request response with the following error message:

```
{"message": "Invalid request body"}
```

This is because the `GET` request that you submitted cannot take a payload and fails the request validation.

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**.

You have now reached the end of this walkthrough.

Build an API Gateway API with HTTP Integrations

To build an API with HTTP integrations, 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

- [Build an API with HTTP Proxy Integration \(p. 39\)](#)
- [Build an API with HTTP Custom Integration \(p. 44\)](#)

Build an API with HTTP Proxy Integration

The HTTP proxy integration of API Gateway 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. In a sense, the HTTP proxy integration makes the client and backend interact directly without any intervention from API Gateway after the API method is set up.

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 \(p. 39\)](#)
- [Test an API with HTTP Proxy Integration \(p. 41\)](#)

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 [Build an API with HTTP Custom Integration \(p. 44\)](#), 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 build an API with HTTP proxy integration with the PetStore website through a proxy resource

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. To create an API, choose **Create new API** (for creating the first API) or **Create API** (for creating any subsequent API). Next, do the following:
 - a. Choose **New API**.
 - b. Type a name in **API Name**.
 - c. Optionally, add a brief description in **Description**.
 - d. Choose **Create API**.

For this tutorial, use `ProxyResourceForPetStore` for the API name.

3. To create a child resource, choose a parent resource item under the **Resources** tree and then choose **Create Resource** from the **Actions** drop-down menu. Then, do the following in the **New Child Resource** pane.

- a. Select the **Configure as proxy resource** option to create a proxy resource. Otherwise, leave it de-selected.
- b. Type a name in the **Resource Name*** input text field.
- c. Type a new name or use the default name in the **Resource Path*** input text field.
- d. Choose **Create Resource**.
- e. Select **Enable API Gateway CORS**, if required.

For this tutorial, select **Configure as proxy resource**. For **Resource Name**, use the default, **proxy**. For **Resource Path**, use **/ {proxy+}**. Select **Enable API Gateway CORS**.

The screenshot shows the 'New Child Resource' creation interface. The URL in the header is: > ProxyResourceForPetStore (miquu3ifg) > Resources > / (7ryftl47g0) > Create. The main title is 'New Child Resource'. Below it, a note says: 'Use this page to create a new child resource for your resource.' There are two main sections: 'Configure as proxy resource' (with a checked checkbox) and 'Resource Name*' (set to 'proxy') and 'Resource Path*' (set to '/ {proxy+}'). A detailed note below these fields explains path parameters and proxy resource behavior. At the bottom, there are 'Enable API Gateway CORS' checkboxes, an 'Actions' dropdown menu (set to 'Resources'), and buttons for 'Cancel' and 'Create Resource'.

4. To set up the **ANY** method for integration with the HTTP backend, do the following:
 - a. Choose the resource just created and then choose **Create Method** from the **Actions** drop-down menu.
 - b. Choose **ANY** from the HTTP method drop-down list and then choose the check mark icon to save the choice.
 - c. Choose **HTTP Proxy for Integration type**.
 - d. Type an HTTP backend resource URL in **Endpoint URL**.
 - e. Use default settings for other fields.
 - f. Choose **Save** to finish configuring the **ANY** method.

For this tutorial, use <http://petstore-demo-endpoint.execute-api.com/{proxy}> for the **Endpoint URL**.

/{{proxy+}} - ANY - Setup



API Gateway will configure your ANY method as a proxy integration. Proxy integrations can communicate with HTTP endpoints or Lambda functions. API Gateway sends the entire request to HTTP endpoints, including resource path, headers, query string parameters, and body. For Lambda integrations, API Gateway applies a default mapping to send all of the request information and responses follow a default interface. To learn more read our [documentation](#)

Integration type Lambda Function Proxy i

HTTP Proxy i

Endpoint URL `demo-endpoint.execute-api.com/{{proxy}}`

Content Handling

Passthrough



Save

In the API just created, the API's proxy resource path of {{proxy+}} becomes 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}}`. The ANY method serves as a placeholder for any of the supported HTTP verbs at run time.

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. To use the API Gateway console to test invoking the API, do the following.
 - a. Choose ANY on a proxy resource in the **Resources** tree.
 - b. Choose Test in the **Method Execution** pane.
 - c. From the **Method** drop-down list, choose an HTTP verb supported by the backend.
 - d. Under **Path**, type a specific path for the proxy resource supporting the chosen operation.
 - e. If required, type a supported query expression for the chosen operation under the **Query Strings** heading.
 - f. If required, type one or more supported header expressions for the chosen operation under the **Headers** heading.
 - g. If configured, set the required stage variable values for the chosen operation under the **Stage Variables** heading.

- h. If prompted and required, choose an API Gateway-generated client certificate under the **Client Certificate** heading to the operation to be authenticated by the back end.
- i. If prompted, type an appropriate request body in the text editor under the **Request Body** heading.
- j. Choose **Test** to test invoking the method.

For this tutorial, use **GET** for **Method** in place of **ANY**, use `petstore/pets` for **Path** in place of the proxy resource path (`{proxy}`), and type=`fish` for **Query Strings**.

[Method Execution](#) /{proxy+} - ANY - Method Test

Make a test call to your method with the provided input

Method

GET

Path

{proxy}

petstore/pets

Query Strings

{proxy}

type=fish

Headers

{proxy}

Use a colon (:) to separate header name and value, and new lines to declare multiple headers. eg.
Accept:application/json.

Stage Variables

No [stage variables](#) exist for this method.

Client Certificate

No client certificates have been generated.

Request Body

Request Body is not supported for GET methods.

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  
}
```

2. To use a browser to call a GET method on a specific resource of the API, do the following.
 - a. If you have not done so, choose **Deploy API** from the **Actions** drop-down menu for the API you created. Follow the instructions to deploy the API to a specific stage. Note the **Invoke URL** that displays on the resulting **Stage Editor** page. This is the base URL of the API.
 - b. To submit a GET request on a specific resource, append the resource path, including possible query string expressions to the **Invoke URL** value obtained in the previous step, copy the complete URL into the address bar of a browser, and choose Enter.

For this tutorial, deploy the API to a `test` stage and append `petstore/pets?type=fish` to the API's Invoke URL. This produces a URL of `https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/petstore/pets?type=fish`.

The result should be the same as returned when you use `TestInvoke` from the API Gateway console.

Build an API with HTTP Custom Integration

In this tutorial, you create an API from ground up 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 the API with HTTP Custom Integration \(p. 45\)](#)

- Map Request Parameters for an API Gateway API (p. 54)
- Map Response Payload (p. 62)

Create the 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.

1. From **Create new API**, select **New API**, type a name in **API Name**, optionally add a description in **Description**, and then choose **Create API**.

Create new API

In Amazon API Gateway, an API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

New API Clone from existing API Import from Swagger Example API

Name and description

Choose a friendly name and description for your API.

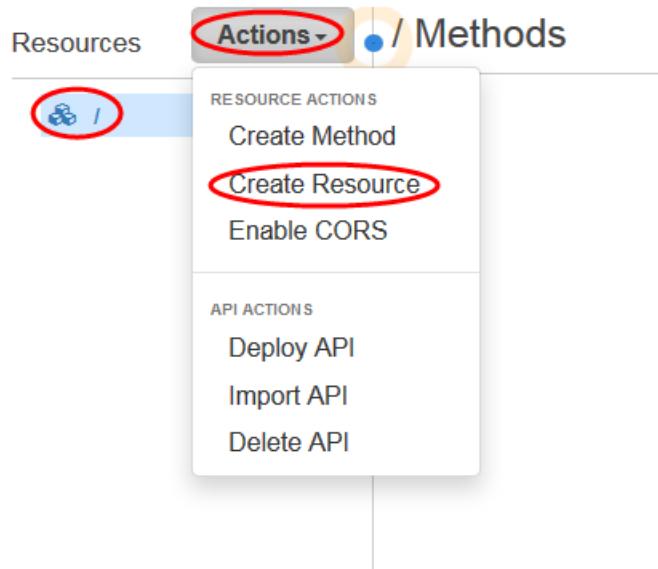
API name* Description

* Required

Create API

As a result, an empty API is created. 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.

2. To create the /pets resource, select the root, choose **Actions** and then choose **Create Resource**.



Type **Pets** in **Resource Name**, leave the **Resource Path** value as given, and choose **Create Resource**.

New Child Resource

Use this page to create a new child resource for your resource.

The 'New Child Resource' creation form. It has two input fields: 'Resource Name*' containing 'Pets' (highlighted with a red circle) and 'Resource Path*' containing '/ pets'. Below the fields is a note: 'You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username''. At the bottom left is a note: '* Required'. On the right are 'Cancel' and 'Create Resource' buttons, with 'Create Resource' highlighted with a red oval.

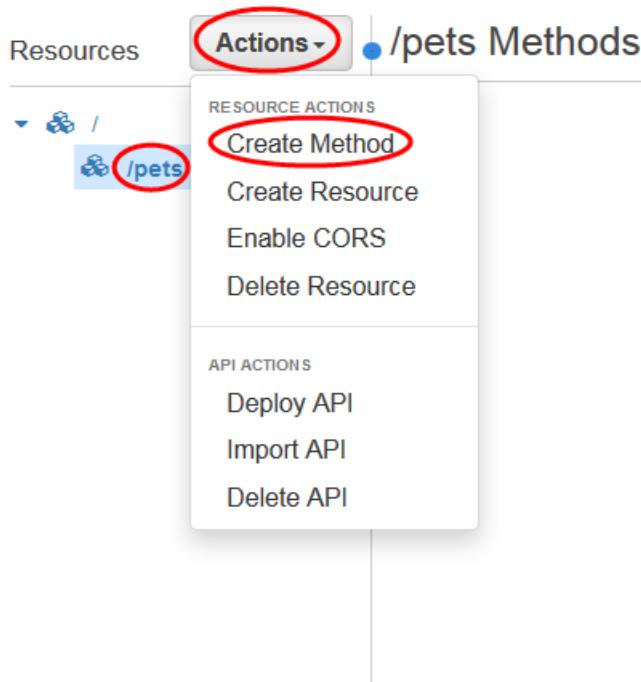
Resource Name*	Pets
Resource Path*	/ pets

You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username'.

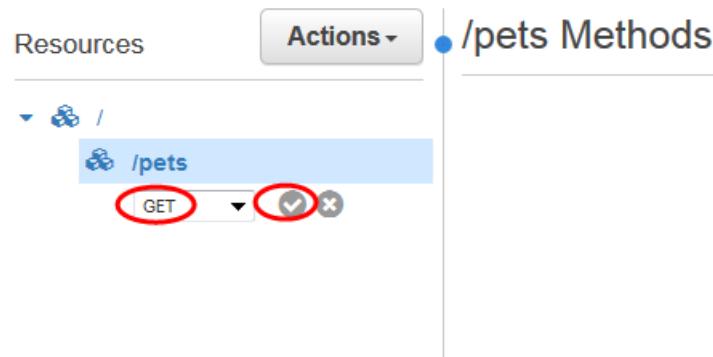
* Required

Cancel Create Resource

3. To expose a GET method on the /pets resource, choose **Actions** and then **Create Method**.



Choose **GET** from the list under the **/pets** resource node and choose the check mark icon to finish creating the method.



Note

Other options for an API method include:

- **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.

The method created is not yet integrated with the backend. The next step sets this up.

4. In the method's **Setup** pane, select **HTTP** for **Integration type**, select **GET** from the **HTTP method** drop-down list, type `http://petstore-demo-endpoint.execute-api.com/petstore/pets` as the **Endpoint URL** value, leave all other settings as default, and then choose **Save**.

Note

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.

/pets - GET - Setup

Choose the integration point for your new method.

Integration type Lambda Function ⓘ
 HTTP ⓘ
 Mock ⓘ
 AWS Service ⓘ

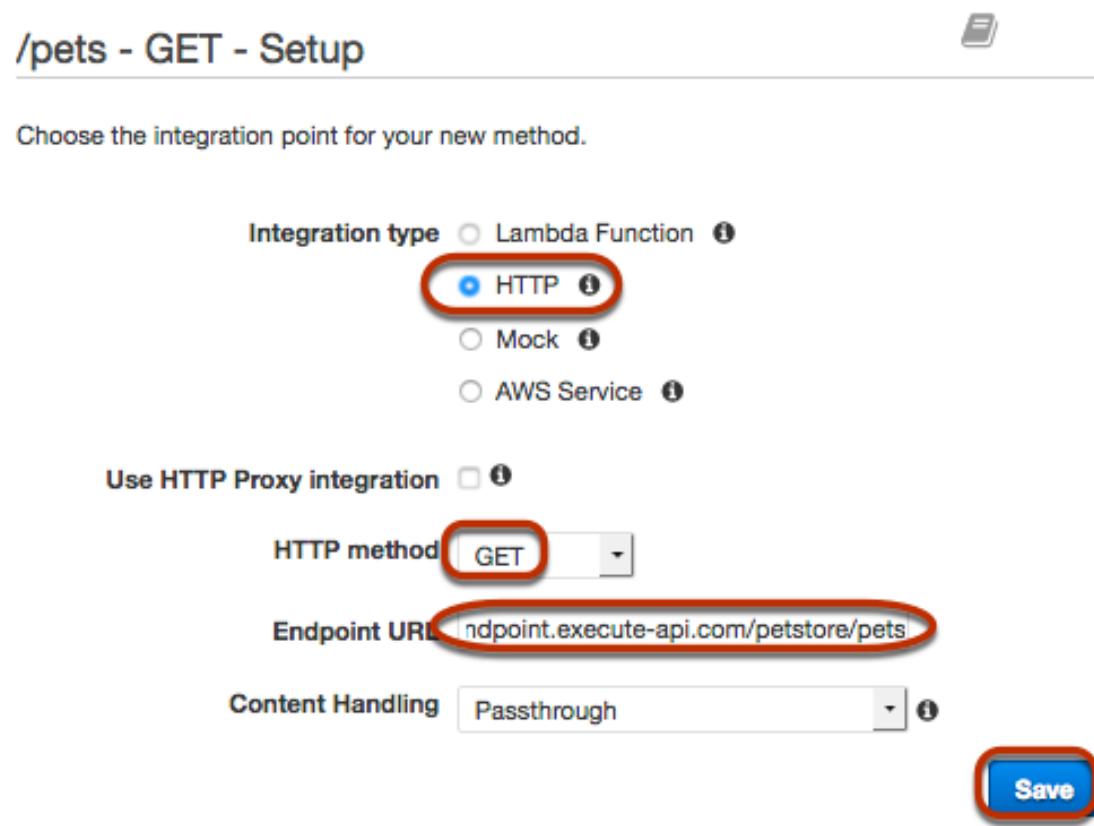
Use HTTP Proxy integration ⓘ

HTTP method GET ⓘ

Endpoint URL `ndpoint.execute-api.com/petstore/pets`

Content Handling Passthrough ⓘ

Save



When the method setup finishes, you are presented with the **Method Execution** pane, where you can further configure the method request to add query string or custom header parameters. You can also update the integration request to map input data from the method request to the format required by the back end.

The PetStore website allows you to retrieve a list of Pet items by the pet type (e.g., "Dog" or "Cat") on a given page. It uses the type and page query string parameters to accept such input. As such, we must add the query string parameters to the method request and map them into the corresponding query strings of the integration request.

5. In the GET method's **Method Execution** pane, choose **Method Request**, select **AWS_IAM** for **Authorization**, expand the **URL Query String Parameters** section, and choose **Add query string** to create two query string parameters named `type` and `page`. Choose the check mark icon to save the newly added query string parameters.

[Method Execution](#) /pets - GET - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Authorization Settings

Authorization **AWS_IAM**  

API Key Required false 

▼ URL Query String Parameters

Name	Caching	
type	<input type="checkbox"/>	
page	<input type="checkbox"/>	

 [Add query string](#)

► HTTP Request Headers

► Request Models [Create a Model](#)

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. Because the method uses AWS_IAM, you must sign the request to invoke the method.

6. From the method's **Integration Request** page, expand the **URL Query String Parameters** section. By default, the method request query string parameters are mapped to the like-named integration request query string parameters. This default mapping works for our demo API. We will leave them as given. To map a different method request parameter to the corresponding integration request parameter, choose the pencil icon for the parameter to edit the mapping expression, shown in the **Mapped from** column. To map a method request parameter to a different integration request parameter, first choose the delete icon to remove the existing integration request parameter, choose **Add query string** to specify a new name and the desired method request parameter mapping expression.

[Method Execution](#) /pets - GET - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function [i](#)

HTTP [i](#)

Mock [i](#)

AWS Service [i](#)

Use HTTP Proxy integration [i](#)

HTTP method GET [e](#)

Endpoint URL <http://petstore-demo-endpoint.execute-api.com/petstore/pets> [e](#)

▶ URL Path Parameters

▼ URL Query String Parameters

Name	Mapped from i	Caching	
type	method.request.querystring.type	<input type="checkbox"/>	e x
page	method.request.querystring.page	<input type="checkbox"/>	e x

[+ Add query string](#)

▶ HTTP Headers

▶ Body Mapping Templates

This completes building the simple demo API. It's time to test the API.

7. To test the API using the API Gateway console, choose **Test** on the **Method Execution** pane for the **GET /pets** method. In the **Method Test** pane, enter **Dog** and **2** for the **type** and **page** query strings, respectively, and then choose **Test**.

[Method Execution](#) /pets - GET - **Method Test**

Make a test call to your method with the provided input

Path

No path parameters exist for this resource. You can define path parameters by using the syntax `{myPathParam}` in a resource path.

Query Strings

type

Dog

page

2

Headers

No header parameters exist for this method. You can add them via Method Request.

Stage Variables

No [stage variables](#) exist for this method.

Client Certificate

No client certificates have been generated.

Request Body

Request Body is not supported for GET methods.



The result is shown as follows. (You may need to scroll down to see the test result.)

Request: /pets?type=Dog&page=2

Status: 200

Latency: 1036 ms

Response Body

```
[  
  {  
    "id": 4,  
    "type": "Dog",  
    "price": 999.99  
  },  
  {  
    "id": 5,  
    "type": "Dog",  
    "price": 249.99  
  },  
  {  
    "id": 6,  
    "type": "Dog",  
    "price": 49.97  
  }  
]
```

Response Headers

```
{"Content-Type": "application/json"}
```

Logs

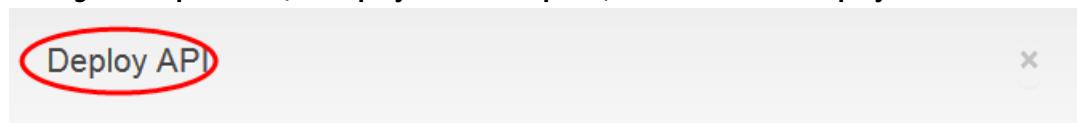
```
Execution log for request test-request  
Mon Apr 04 05:48:01 UTC 2016 : Starting execution for request: test-invoke-request  
Mon Apr 04 05:48:01 UTC 2016 : HTTP Method: GET, Resource Path: /pets  
Mon Apr 04 05:48:01 UTC 2016 : Method request path: {}  
Mon Apr 04 05:48:01 UTC 2016 : Method request query string: {page=2, type=Dog}  
Mon Apr 04 05:48:01 UTC 2016 : Method request headers: {}  
Mon Apr 04 05:48:01 UTC 2016 : Method request body before transformations: null
```

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

8. To deploy the API, select the API and then choose **Deploy API** from the **Actions** drop-down menu.

The screenshot shows the left sidebar with 'myApi' selected. The main area displays the 'Resources' section with a tree view of '/'. A context menu is open over the root resource, with 'Actions' selected. The 'API ACTIONS' section contains 'Deploy API', which is circled in red.

In the **Deploy API** dialog, choose a stage (or [New Stage] for the API's first deployment); enter a name (e.g., "test", "prod", "dev", etc.) in the **Stage name** input field; optionally, provide a description in **Stage description** and/or **Deployment description**; and then choose **Deploy**.



Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

The dialog box now shows the filled-in fields: 'Deployment stage' is '[New Stage]', 'Stage name*' is 'test', 'Stage description' is 'GET on Pets only', and 'Deployment description' is 'Initial deployment'. The 'Deploy' button at the bottom right is circled in red.

Once deployed, you can obtain the invocation URLs (**Invoke URL**) of the API's endpoints.

If the GET method supported open access, (i.e., if the method's authorization type were set to **NONE**) you could double-click the **Invoke URL** link to invoke the method in your default browser. If

needed, you could also append necessary query string parameters to the invocation URL. With the **AWS_IAM** authorization type described here, you must sign the request with an [access key ID and the corresponding secret key](#) of an IAM user of your AWS account. To do this, you must use a client that supports the [Signature Version 4](#) (SigV4) protocols. An example of such a client is an app that uses one of the AWS SDKs or the [Postman](#) application or cURL commands. To call a POST, PUT, or PATCH method that take a payload, you also need to use such a client to handle the payload.

To invoke this API method in the Postman, append the query string parameters to the stage-specific method invocation URL (as shown in the previous image) to create the complete method request URL:

```
https://api-id.execute-api.region.amazonaws.com/test/pets?type=Dog&page=2
```

Specify this URL in the address bar of the browser. Choose **GET** as the HTTP verb. Select **AWS Signature** for the **Type** option under the **Authorization** tab, and then specify the following required properties before sending the request:

- For **AccessKey**, type the caller's AWS access key, as provisioned from [AWS IAM](#).
- For **SecretKey**, type the caller's AWS secret key, as provisioned from AWS IAM when the access key was first created.
- For **AWS Region**, type the API-hosting AWS Region, as specified in the invocation URL.
- For **Service Name**, type `execute-api`, for the API Gateway execution service.

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.

Map Request Parameters for an API Gateway API

In this walkthrough, we describe how to map method request parameters to the corresponding integration request parameters for an API Gateway API. We create an example API with the HTTP custom integration and use it to demonstrate how to use API Gateway to map a method request parameter to the corresponding integration request parameter. We then access the following publicly accessible HTTP endpoint:

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets
```

If you copy the above URL, paste it into the address bar of a web browser, and press **Enter** or **Return**, you get the following JSON-formatted response body:

```
[  
  {  
    "id": 1,  
    "type": "dog",  
    "price": 249.99  
  },  
  {  
    "id": 2,  
    "type": "cat",  
    "price": 124.99  
  },  
  {  
    "id": 3,
```

```
[  
    {"type": "fish",  
     "price": 0.99  
    }  
]
```

The preceding endpoint can take two query parameters: `type` and `page`. For example, change the URL to the following:

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets?type=cat&page=2
```

You receive the following JSON-formatted response payload, displaying page 2 of only the cats:

```
[  
    {  
        "id": 4,  
        "type": "cat",  
        "price": 999.99  
    },  
    {  
        "id": 5,  
        "type": "cat",  
        "price": 249.99  
    },  
    {  
        "id": 6,  
        "type": "cat",  
        "price": 49.97  
    }  
]
```

This endpoint also supports the use of an item ID, as expressed by a URL path parameter. For example, browse to the following:

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets/1
```

The following JSON-formatted information about the item with an ID of 1 is displayed:

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

In addition to supporting GET operations, this endpoint takes POST requests with a payload. For example, use [Postman](#) to send a POST method request to the following:

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets
```

Include the header `Content-type: application/json` and the following request body:

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

You receive the following JSON object in the response body:

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

We now expose these and other features by building an API Gateway API with the HTTP custom integration of this PetStore website. The tasks include the following:

- Create an API with a resource of `https://my-api-id.execute-api.region-id.amazonaws.com/test/petstorewalkthrough/pets` acting as a proxy to the HTTP endpoint of `http://petstore-demo-endpoint.execute-api.com/petstore/pets`.
- Enable the API to accept two method request query parameters of `petType` and `petsPage`, map them to the `type` and `page` query parameters of the integration request, respectively, and pass the request to the HTTP endpoint.
- Support 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.
- Enable the method request to accept the JSON payload of the format defined by the backend website, and pass the payload without modification through the integration request to the backend HTTP endpoint.

Topics

- [Prerequisites \(p. 56\)](#)
- [Step 1: Create Resources \(p. 56\)](#)
- [Step 2: Create and Test the Methods \(p. 57\)](#)
- [Step 3: Deploy the API \(p. 60\)](#)
- [Step 4: Test the API \(p. 60\)](#)
- [Next Steps \(p. 62\)](#)

Note

Pay attention to the casing used in the steps of this walkthrough. Typing a lowercase letter instead of an uppercase letter (or vice versa) can cause errors later in the walkthrough.

Prerequisites

Before you begin this walkthrough, you should do the following:

1. Complete the steps in [Get Ready to Build an API Gateway API \(p. 6\)](#), including assigning API Gateway access permission to the IAM user.
2. At a minimum, follow the steps in [Build an API with HTTP Custom Integration \(p. 44\)](#) to create a new API named `MyDemoAPI` in the API Gateway console.

Step 1: Create Resources

In this step, you create three resources that enable the API to interact with the HTTP endpoint.

To create the first resource

1. In the **Resources** pane, select the resource root, as represented by a single forward slash (/), and then choose **Create Resource** from the **Actions** drop-down menu.

2. For **Resource Name**, type **petstorewalkthrough**.
3. For **Resource Path**, accept the default of **/petstorewalkthrough**, and then choose **Create Resource**.

To create the second resource

1. In the **Resources** pane, choose **/petstorewalkthrough**, and then choose **Create Resource**.
2. For **Resource Name**, type **pets**.
3. For **Resource Path**, accept the default of **/petstorewalkthrough/pets**, and then choose **Create Resource**.

To create the third resource

1. In the **Resources** pane, choose **/petstorewalkthrough/pets**, and then choose **Create Resource**.
2. For **Resource Name**, type **petId**. This maps to the item ID in the HTTP endpoint.
3. For **Resource Path**, overwrite **petId** with **{petId}**. Use curly braces (**{ }**) around **petId** so that **/petstorewalkthrough/pets/{petId}** is displayed, and then choose **Create Resource**.

This maps to **/petstore/pets/*my-item-id*** in the HTTP endpoint.

Step 2: Create and Test the Methods

In this step, you integrate the methods with the backend HTTP endpoints, map the GET method request parameters to the corresponding integration request parameters, and then test the methods.

To set up and test the first GET method

This procedure demonstrates the following:

- Create and integrate the method request of **GET /petstorewalkthrough/pets** with the integration request of **GET http://petstore-demo-endpoint.execute-api.com/petstore/pets**.
- Map the method request query parameters of **petType** and **petsPage** to the integration request query string parameters of **type** and **page**, respectively.

1. In the **Resources** pane, choose **/petstorewalkthrough/pets**, choose **Create Method** from the **Actions** menu, and then choose **GET** under **/pets** from the drop-down list of the method names.
2. In the **/petstorewalkthrough/pets - GET - Setup** pane, choose **HTTP** for **Integration type** and choose **GET** for **HTTP method**.
3. For **Endpoint URL**, type **http://petstore-demo-endpoint.execute-api.com/petstore/pets**.
4. Choose **Save**.
5. In the **Method Execution** pane, choose **Method Request**, and then choose the arrow next to **URL Query String Parameters**.
6. Choose **Add query string**.
7. For **Name**, type **petType**.

This specifies the **petType** query parameter in the API's method request.

8. Choose the check mark icon to finish creating the method request URL query string parameter.
9. Choose **Add query string** again.
10. For **Name**, type **petsPage**.

This specifies the `petsPage` query parameter in the API's method request.

11. Choose the check mark icon to finish creating the method request URL query string parameter.
12. Choose **Method Execution**, choose **Integration Request**, and then choose the arrow next to **URL Query String Parameters**.
13. Delete the `petType` entry mapped from `method.request.querystring.petType` and the `petsPage` entry mapped from `method.request.querystring.petsPage`. You perform this step because the endpoint requires query string parameters named `type` and `page` for the request URL, instead of the default values.
14. Choose **Add query string**.
15. For **Name**, type `type`. This creates the required query string parameter for the integration request URL.
16. For **Mapped from**, type `method.request.querystring.petType`.

This maps the method request's `petType` query parameter to the integration request's `type` query parameter.

17. Choose the check mark icon to finish creating the integration request URL query string parameter.
18. Choose **Add query string** again.
19. For **Name**, type `page`. This creates the required query string parameter for the integration request URL.
20. For **Mapped from**, type `method.request.querystring.petsPage`.

This maps the method request's `petsPage` query parameter to the integration request's `page` query parameter.

21. Choose the check mark icon to finish creating the integration request URL query string parameter.
22. Choose **Method Execution**. In the **Client** box, choose **TEST**. In the **Query Strings** area, for **petType**, type `cat`. For **petsPage**, type `2`.
23. Choose **Test**. If successful, **Response Body** displays the following:

```
[
  {
    "id": 4,
    "type": "cat",
    "price": 999.99
  },
  {
    "id": 5,
    "type": "cat",
    "price": 249.99
  },
  {
    "id": 6,
    "type": "cat",
    "price": 49.97
  }
]
```

To set up and test the second GET method

This procedure demonstrates the following:

- Create and integrate the method request of `GET /petstorewalkthrough/pets/{petId}` with the integration request of `GET http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`.

- Map the method request path parameters of `petId` to the integration request path parameters of `id`.
1. In the **Resources** list, choose `/petstorewalkthrough/pets/{petId}`, choose **Create Method** from the **Actions** drop-down menu, and then choose **GET** as the HTTP verb for the method.
 2. In the **Setup** pane, choose **HTTP** for **Integration type** and choose **GET** for **HTTP method**.
 3. For **Endpoint URL**, type `http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`.
 4. Choose **Save**.
 5. In the **Method Execution** pane, choose **Integration Request**, and then choose the arrow next to **URL Path Parameters**.
 6. Choose **Add path**.
 7. For **Name**, type `id`.
 8. For **Mapped from**, type `method.request.path.petId`.

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

9. Choose the check mark icon to finish creating the URL path parameter.
10. Choose **Method Execution**, and in the **Client** box, choose **TEST**. In the **Path** area, for **petId**, type `1`.
11. Choose **Test**. If successful, **Response Body** displays the following:

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

To set up and test the POST method

This procedure demonstrates the following:

- Create and integrate the method request of `POST /petstorewalkthrough/pets` with the integration request of `POST http://petstore-demo-endpoint.execute-api.com/petstore/pets`.
- Pass the method request JSON payload through to the integration request payload, without modification.

1. In the **Resources** pane, choose `/petstorewalkthrough/pets`, choose **Create Method** from the **Actions** drop-down menu, and then choose **POST** as the HTTP verb for the method.
2. In the **Setup** pane, choose **HTTP** for **Integration type** and choose **POST** for **HTTP method**.
3. For **Endpoint URL**, type `http://petstore-demo-endpoint.execute-api.com/petstore/pets`.
4. Choose **Save**.
5. In the **Method Execution** pane, in the **Client** box, choose **TEST**. Expand **Request Body**, and then type the following:

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

6. Choose **Test**. If successful, **Response Body** displays the following:

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

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. In the **Resources** pane, choose **Deploy API**.
2. For **Deployment stage**, choose **test**.

Note

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

3. For **Deployment description**, type **Calling HTTP endpoint walkthrough**.
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 **Stage Editor** pane, next to **Invoke URL**, copy the URL to the clipboard. It should look something like this:

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

2. Paste this URL in the address box of a new browser tab.
3. Append /petstorewalkthrough/pets so that it looks like this:

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

Browse to the URL. The following information should be displayed:

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

]

4. After `petstorewalkthrough/pets`, type `?petType=cat&petsPage=2` so that it looks like this:

`https://my-api-id.execute-api.region-id.amazonaws.com/test/petstorewalkthrough/pets?petType=cat&petsPage=2`

5. Browse to the URL. The following information should be displayed:

```
[  
  {  
    "id": 4,  
    "type": "cat",  
    "price": 999.99  
  },  
  {  
    "id": 5,  
    "type": "cat",  
    "price": 249.99  
  },  
  {  
    "id": 6,  
    "type": "cat",  
    "price": 49.97  
  }  
]
```

6. After `petstorewalkthrough/pets`, replace `?petType=cat&petsPage=2` with `/1` so that it looks like this:

`https://my-api-id.execute-api.region-id.amazonaws.com/test/petstorewalkthrough/pets/1`

7. Browse to the URL. The following information should be displayed:

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

8. Using a web debugging proxy tool or the cURL command-line tool, send a POST method request to the URL from the previous procedure. Append `/petstorewalkthrough/pets` so that it looks like this:

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

Append the following header:

`Content-Type: application/json`

Add the following code to the request body:

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

For example, if you use the cURL command-line tool, run a command similar to the following:

```
curl -H "Content-Type: application/json" -X POST -d "{\"type\": \"dog\", \"price\": 249.99}" https://my-api-id.execute-api.region-id.amazonaws.com/test/petstorewalkthrough/pets
```

The following information should be returned in the response body:

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

You have reached the end of this walkthrough.

Next Steps

The next walkthrough shows how to use models and mappings in API Gateway to transform the output of an API call from one data format to another. See [Map Response Payload \(p. 62\)](#).

Map Response Payload

In this walkthrough, we show how to use models and mapping templates in API Gateway to transform the output of an API call from one data schema to another. This walkthrough builds on the instructions and concepts in the [Getting Started with Amazon API Gateway \(p. 6\)](#) and the [Map Request Parameters for an API Gateway API \(p. 54\)](#). If you have not yet completed those walkthroughs, we suggest you do them first.

This walkthrough uses API Gateway to get example data from a publicly accessible HTTP endpoint and from an AWS Lambda function you create. Both the HTTP endpoint and the Lambda function return the same example data:

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

You will use models and mapping templates to transform this data to one or more output formats. In API Gateway, a model defines the format, also known as the schema or shape, of some data. In API Gateway, a mapping template is used to transform some data from one format to another. For more information, see [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

The first model and mapping template is used to rename `id` to `number`, `type` to `class`, and `price` to `salesPrice`, as follows:

```
[  
  {  
    "number": 1,  
    "class": "dog",  
    "salesPrice": 249.99  
  },  
  {  
    "number": 2,  
    "class": "cat",  
    "salesPrice": 124.99  
  },  
  {  
    "number": 3,  
    "class": "fish",  
    "salesPrice": 0.99  
  }  
]
```

The second model and mapping template is used to combine `id` and `type` into `description`, and to rename `price` to `askingPrice`, as follows:

```
[  
  {  
    "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 third model and mapping template is used to combine `id`, `type`, and `price` into a set of `listings`, as follows:

```
{  
  "listings": [  
    "Item 1 is a dog. The asking price is 249.99.",  
    "Item 2 is a cat. The asking price is 124.99.",  
    "Item 3 is a fish. The asking price is 0.99."  
  ]  
}
```

Topics

- [Step 1: Create Models \(p. 64\)](#)
- [Step 2: Create Resources \(p. 65\)](#)
- [Step 3: Create GET Methods \(p. 66\)](#)
- [Step 4: Create a Lambda Function \(p. 67\)](#)
- [Step 5: Set up and Test the Methods \(p. 68\)](#)
- [Step 6: Deploy the API \(p. 71\)](#)
- [Step 7: Test the API \(p. 71\)](#)
- [Step 8: Clean Up \(p. 73\)](#)

- [Next Steps \(p. 73\)](#)

Step 1: Create Models

In this step, you create four models. The first three models represent the data output formats for use with the HTTP endpoint and the Lambda function. The last model represents the data input schema for use with the Lambda function.

To create the first output model

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. If **MyDemoAPI** is displayed, choose **Models**.
3. Choose **Create**.
4. For **Model name**, type **PetsModelNoFlatten**.
5. For **Content type**, type **application/json**.
6. For **Model description**, type **Changes id to number, type to class, and price to salesPrice**.
7. For **Model schema**, type the following JSON Schema-compatible definition:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "PetsModelNoFlatten",  
  "type": "array",  
  "items": {  
    "type": "object",  
    "properties": {  
      "number": { "type": "integer" },  
      "class": { "type": "string" },  
      "salesPrice": { "type": "number" }  
    }  
  }  
}
```

8. Choose **Create model**.

To create the second output model

1. Choose **Create**.
2. For **Model name**, type **PetsModelFlattenSome**.
3. For **Content type**, type **application/json**.
4. For **Model description**, type **Combines id and type into description, and changes price to askingPrice**.
5. For **Model schema**, type the following:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "PetsModelFlattenSome",  
  "type": "array",  
  "items": {  
    "type": "object",  
    "properties": {  
      "description": { "type": "string" },  
      "askingPrice": { "type": "number" }  
    }  
  }  
}
```

}

6. Choose **Create model**.

To create the third output model

1. Choose **Create**.
2. For **Model name**, type **PetsModelFlattenAll**.
3. For **Content type**, type **application/json**.
4. For **Model description**, type **Combines id, type, and price into a set of listings**.
5. For **Model schema**, type the following:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "PetsModelFlattenAll",  
  "type": "object",  
  "properties": {  
    "listings": {  
      "type": "array",  
      "items": {  
        "type": "string"  
      }  
    }  
  }  
}
```

6. Choose **Create model**.

To create the input model

1. Choose **Create**.
2. For **Model name**, type **PetsLambdaModel**.
3. For **Content type**, type **application/json**.
4. For **Model description**, type **GetPetsInfo model**.
5. For **Model schema**, type the following:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "PetsLambdaModel",  
  "type": "array",  
  "items": {  
    "type": "object",  
    "properties": {  
      "id": { "type": "integer" },  
      "type": { "type": "string" },  
      "price": { "type": "number" }  
    }  
  }  
}
```

6. Choose **Create model**.

Step 2: Create Resources

In this step, you create four resources. The first three resources enable you to get the example data from the HTTP endpoint in the three output formats. The last resource enables you to get the example data

from the Lambda function in the output schema that combines `id` and `type` into `description` and renames `price` to `askingPrice`.

To create the first resource

1. In the links list, choose **Resources**.
2. In the **Resources** pane, choose `/petstorewalkthrough`, and then choose **Create Resource**.
3. For **Resource Name**, type `NoFlatten`.
4. For **Resource Path**, accept the default of `/petstorewalkthrough/noflatten`, and then choose **Create Resource**.

To create the second resource

1. In the **Resources** pane, choose `/petstorewalkthrough` again, and then choose **Create Resource**.
2. For **Resource Name**, type `FlattenSome`.
3. For **Resource Path**, accept the default of `/petstorewalkthrough/flattensome`, and then choose **Create Resource**.

To create the third resource

1. In the **Resources** pane, choose `/petstorewalkthrough` again, and then choose **Create Resource**.
2. For **Resource Name**, type `FlattenAll`.
3. For **Resource Path**, accept the default of `/petstorewalkthrough/flattenall`, and then choose **Create Resource**.

To create the fourth resource

1. In the **Resources** pane, choose `/petstorewalkthrough` again, and then choose **Create Resource**.
2. For **Resource Name**, type `LambdaFlattenSome`.
3. For **Resource Path**, accept the default of `/petstorewalkthrough/lambdaflattensome`, and then choose **Create Resource**.

Step 3: Create GET Methods

In this step, you create a GET method for each of the resources you created in the previous step.

To create the first GET method

1. In the **Resources** list, choose `/petstorewalkthrough/flattenall`, and then choose **Create Method**.
2. From the drop-down list, choose **GET**, and then choose the check mark icon to save your choice.
3. In the **Setup** pane, choose **HTTP** for the **Integration type** and **GET** for **HTTP method**, type `http://petstore-demo-endpoint.execute-api.com/petstore/pets` in **Endpoint URL**, and choose **Save**.

To create the second GET method

1. In the **Resources** list, choose `/petstorewalkthrough/lambdaflattensome`, and then choose **Create Method**.
2. From the drop-down list, choose **GET**, and then choose the check mark to save your choice.
3. In the **Setup** pane, choose **Lambda Function** for the **Integration type**, choose the region where you have created the [GetPetsInfo Lambda function \(p. 67\)](#) from the **Lambda Region** drop-down

list, choose **GetPetsInfo** for **Lambda Function**, and choose **Save**. Choose **OK** when prompted to add permission to the Lambda function.

To create the third GET method

1. In the **Resources** list, choose **/petstorewalkthrough/flattensome**, and then choose **Create Method**.
2. From the drop-down list, choose **GET**, and then choose the check mark icon to save your choice.
3. In the Setup pane, choose **HTTP** for the **Integration type** and **GET** for **HTTP method**, type **http://petstore-demo-endpoint.execute-api.com/petstore/pets** in **Endpoint URL**, and then choose **Save**.

To create the fourth GET method

1. In the **Resources** list, choose **/petstorewalkthrough/noflatten**, and then choose **Actions**, **Create Method**.
2. From the drop-down list, choose **GET**, and then choose the check mark icon to save your choice.
3. In the Setup pane, choose **HTTP** for the **Integration type** and **GET** for **HTTP method**, type **http://petstore-demo-endpoint.execute-api.com/petstore/pets** in **Endpoint URL**, and then choose **Save**.

Step 4: Create a Lambda Function

In this step, you create a Lambda function that returns the sample data.

To create the Lambda function

1. Open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Do one of the following:
 - If a welcome page appears, choose **Get Started Now**.
 - If the **Lambda: Function list** page appears, choose **Create a Lambda function**.
3. For **Name**, type **GetPetsInfo**.
4. For **Description**, type **Gets information about pets**.
5. For **Code template**, choose **None**.
6. Type the following code:

```
console.log('Loading event');

exports.handler = function(event, context, callback) {
    callback(null,
        [{"id": 1, "type": "dog", "price": 249.99},
         {"id": 2, "type": "cat", "price": 124.99},
         {"id": 3, "type": "fish", "price": 0.99}]); // SUCCESS with message
};
```

Tip

In the preceding code, written in Node.js, `console.log` writes information to an Amazon CloudWatch log. `event` contains the input to the Lambda function. `context` contains calling context. `callback` returns the result (for Node.js 4.3 and later). For more information about how to write Lambda function code, see the "Programming Model" section in [AWS Lambda: How it Works](#) and the sample walkthroughs in the [AWS Lambda Developer Guide](#).

7. For **Handler name**, leave the default of `index.handler`.
8. For **Role**, choose the Lambda execution role, **APIGatewayLambdaExecRole**, you created in the [Build an API Gateway API with Lambda Integration \(p. 18\)](#).
9. Choose **Create Lambda function**.
10. In the list of functions, choose **GetPetsInfo** to show the function's details.
11. Make a note of the AWS region where you created this function. You need it later.
12. In the pop-up list, choose **Edit or test function**.
13. For **Sample event**, replace any code that appears with the following:

```
{
}
```

Tip

The empty curly braces mean that there are no input values for this Lambda function. This function simply returns the JSON object containing the pets information, so those key-value pairs are not required here.

14. Choose **Invoke**. **Execution result** shows `[{"id":1,"type":"dog","price":249.99}, {"id":2,"type":"cat","price":124.99}, {"id":3,"type":"fish","price":0.99}]`, which is also written to the CloudWatch Logs log files.
15. Choose **Go to function list**.

Step 5: Set up and Test the Methods

In this step, you configure the method responses, integration requests, and integration responses to specify the input and output data schemas (or models) for the GET methods associated with the HTTP endpoint and the Lambda function. You also learn to test calling these methods using the API Gateway console.

To set up the integration for the first GET method and then test it

1. From the API's **Resources** tree, choose **GET** under the `/petstorewalkthrough/flattenall` node.
2. In the **Method Execution** pane, choose **Method Response**, and then choose the arrow next to **200**.
3. In the **Response Models for 200** area, for **application/json**, choose the pencil icon to start setting up the model for the method output. For **Models**, choose **PetsModelFlattenAll**, and then choose the check mark icon to save the setting.
4. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200**.
5. Expand the **Body Mapping Templates** section, choose **application/json** under **Content-Type**.
6. For **Generate template from model**, choose **PetsModelFlattenAll** to display a mapping template after the `PetsModelFlattenAll` model as a starting point.
7. Modify the mapping template code as follows:

```
#set($inputRoot = $input.path('$'))
{
    "listings" : [
#foreach($elem in $inputRoot)
        "Item number $elem.id is a $elem.type. The asking price is
$elem.price."#if($foreach.hasNext),#end

#end
    ]
}
```

8. Choose **Save**.
9. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** displays the following:

```
{
  "listings" : [
    "Item number 1 is a dog. The asking price is 249.99.",
    "Item number 2 is a cat. The asking price is 124.99.",
    "Item number 3 is a fish. The asking price is 0.99."
  ]
}
```

To set up integration for the second GET method and then test it

1. From the API's **Resources** tree, choose **GET** under the **/petstorewalkthrough/lambdaflattensome** node.
2. In **Method Execution**, choose **Method Response**. And then choose the arrow next to **200** to expand the section.
3. In the **Response Models for 200** area, choose the pencil icon on the row for the content type of **application/json**. Choose **PetsModelFlattenSome** for **Models**, and then choose the check mark icon to save the choice.
4. Go back to **Method Execution**. Choose **Integration Response**, and then choose the arrow next to **200**.
5. In the **Body Mapping Templates** section, choose **application/json** under **Content-Type**.
6. For **Generate template**, choose **PetsModelFlattenSome** to display the mapping script template for the output of this method.
7. Modify the code as follows, and then choose **Save**:

```
#set($inputRoot = $input.path(''))
[
#foreach($elem in $inputRoot)
{
  "description" : "Item $elem.id is a $elem.type.",
  "askingPrice" : $elem.price
}#if($foreach.hasNext),#end

#end
]
```

8. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** displays the following:

```
[
  {
    "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 set up integration for the third GET method and then test it

1. From the API's **Resources** tree, choose **GET** under the **/petstorewalkthrough/flattensome** node.
2. In the **Method Execution** pane, choose **Method Response**.
3. Choose the arrow next to **200**.
4. In the **Response Models for 200** area, for **application/json**, choose the pencil icon. For **Models**, choose **PetsModelFlattenSome**, and then choose the check-mark icon to save the choice.
5. Go back to **Method Execution** and choose **Integration Response**.
6. Choose the arrow next to **200** to expand the section.
7. Expand the **Body Mapping Templates** area. Choose **application/json** for **Content-Type**. For **Generate template**, choose **PetsModelFlattenSome** to display a mapping script template for the output of this method.
8. Modify the code as follows:

```
#set($inputRoot = $input.path('$'))  
[  
#foreach($elem in $inputRoot)  
{  
    "description": "Item $elem.id is a $elem.type.",  
    "askingPrice": $elem.price  
}#if($foreach.hasNext),#end  
  
#end  
]
```

9. Choose **Save**.
10. Go back to **Method Execution** and choose **TEST** in the **Client** box. And then choose **Test**. If successful, **Response Body** displays the following:

```
[  
{  
    "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 set up integration for the fourth GET method and then test it

1. From the API's **Resources** tree, choose **GET** under the **/petstorewalkthrough/noflatten** node.
2. In the **Method Execution** pane, choose **Method Response**, and then expand the **200** section.
3. In the **Response Models for 200** area, for **application/json**, choose the pencil icon to update the response model for this method.
4. Choose **PetsModelNoFlatten** as the model for the content type of **application/json**, and then choose the check-mark icon to save the choice.
5. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200** to expand the section.

6. Expand the **Mapping Templates** section. Choose **application/json** for **Content-Type**. For **Generate templates**, choose **PetsModelNoFlatten** to display a mapping script template for the output of this method.
7. Modify the code as follows:

```
#set($inputRoot = $input.path('$'))  
[  
#foreach($elem in $inputRoot)  
{  
    "number": $elem.id,  
    "class": "$elem.type",  
    "salesPrice": $elem.price  
}#if($foreach.hasNext),#end  
  
#end  
]
```

8. Choose **Save**.
9. Go back to **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** displays the following:

```
[  
{  
    "number": 1,  
    "class": "dog",  
    "salesPrice": 249.99  
,  
{  
    "number": 2,  
    "class": "cat",  
    "salesPrice": 124.99  
,  
{  
    "number": 3,  
    "class": "fish",  
    "salesPrice": 0.99  
}  
]
```

Step 6: 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. In the **Resources** pane, choose **Deploy API**.
2. For **Deployment stage**, choose **test**.
3. For **Deployment description**, type **Using models and mapping templates walkthrough**.
4. Choose **Deploy**.

Step 7: Test the API

In this step, you go outside of the API Gateway console to interact with both the HTTP endpoint and the Lambda function.

1. In the **Stage Editor** pane, next to **Invoke URL**, copy the URL to the clipboard. It should look something like this:

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

2. Paste this URL in the address box of a new browser tab.
3. Append /petstorewalkthrough/noflatten so that it looks like this:

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

Browse to the URL. The following information should be displayed:

```
[  
  {  
    "number": 1,  
    "class": "dog",  
    "salesPrice": 249.99  
  },  
  {  
    "number": 2,  
    "class": "cat",  
    "salesPrice": 124.99  
  },  
  {  
    "number": 3,  
    "class": "fish",  
    "salesPrice": 0.99  
  }  
]
```

4. After petstorewalkthrough/, replace noflatten with flattensome.
5. Browse to the URL. The following information should be displayed:

```
[  
  {  
    "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  
  }  
]
```

6. After petstorewalkthrough/, replace flattensome with flattenall.
7. Browse to the URL. The following information should be displayed:

```
{  
  "listings" : [  
    "Item number 1 is a dog. The asking price is 249.99.",  
    "Item number 2 is a cat. The asking price is 124.99.",  
    "Item number 3 is a fish. The asking price is 0.99."  
  ]  
}
```

8. After petstorewalkthrough/, replace flattenall with lambdaflattensome.
9. Browse to the URL. The following information should be displayed:

```
[  
  {  
    "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  
  }  
]
```

Step 8: Clean Up

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

Warning

If you delete a Lambda function 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 a Lambda function relies on, the Lambda function 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 the resource. If you plan to continue experimenting with the resources you created for this and the other walkthroughs, do not delete the Lambda invocation role or the Lambda execution role.

API Gateway does not currently support the deactivation or deletion of APIs that no longer work.

To delete the Lambda function

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. On the **Lambda: Function list** page, in the list of functions, choose the button next to **GetPetsInfo**, and then choose **Actions, Delete**. When prompted, choose **Delete** again.

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 **APIGatewayLambdaExecRole**, and then choose **Role Actions, Delete Role**. When prompted, choose **Yes, Delete**.
4. In the **Details** area, choose **Policies**.
5. Select **APIGatewayLambdaExecPolicy**, and then choose **Policy Actions, Delete**. When prompted, choose **Delete**.

You have now reached the end of this walkthrough.

Next Steps

You may want to begin the next walkthrough, which shows you how to create an API Gateway API to access an AWS service. See [Build an API Gateway API with AWS Integration \(p. 75\)](#).

Build an 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 or choose an existing [VpcLink](#) connected to a network load balancer that targets the desired VPC resources. You must have [appropriate permissions \(p. 147\)](#) 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](#).

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 \(p. 147\)](#).
2. Grant permissions to create the VPC links for private integrations. For more information, see [Grant Permissions to Create a VPC Link \(p. 147\)](#).

After creating your VPC resource and your network load balancer with your VPC resource configured in its target groups, use the following 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 private integration using the API Gateway console

1. Sign in to the API Gateway console and choose a region; for example, `us-west-2`, on the navigation bar.
2. Create a VPC link, if you have not already done so:
 - a. From the primary navigation pane, choose **VPC Links** and then choose **+ Create**.
 - b. Under **VPC Link**, type a name (for example, `my-test-vpc-link`) in the **Name** field.
 - c. Optionally, give a description of the VPC link in the **Description** text area.
 - d. Choose a network load balancer from the **Target NLB** drop-down list.

You must have the network load balancer already created in the region you chose for the network load balancer to be present in the list.

- e. Choose **Create** to start creating the VPC link.

The initial response returns a [VpcLink](#) resource representation with the VPC link ID and a `PENDING` status. This is because the operation is asynchronous and takes about 2-4 minutes to complete. Upon successful completion, the status is `AVAILABLE`. In the meantime, you can proceed to create the API.

3. Choose **APIs** from the primary navigation pane and then choose **+ Create API** to create a new API of either an edge-optimized or regional endpoint type.

4. For the root resource (/), choose **Create Method** from the **Actions** drop-down menu, and then choose GET.
5. In the **/ GET - Setup** pane, initialize the API method integration as follows:
 - a. Choose VPC Link for **Integration type**.
 - b. Choose **Use Proxy Integration**.
 - c. From the **Method** drop-down list, choose GET as the integration method.
 - d. From the **VPC Link** drop-down list, choose [Use Stage Variables] and type \${stageVariables.vpcLinkId} in the text box below.

We will define the vpcLinkId stage variable after deploying the API to a stage and set its value to the ID of the VpcLink created in **Step 1**.
- e. Type a URL, for example, `http://myApi.example.com`, for **Endpoint URL**.

Here, the host name (for example, myApi.example.com) is used to set the Host header of the integration request.
- f. Leave the **Use Default Timeout** selection as-is, unless you want to customize the integration timeouts.
- g. Choose **Save** to finish setting up the integration.

With the proxy integration, the API is ready for deployment. Otherwise, you need to proceed to set up appropriate method responses and integration responses.
- h. From the **Actions** drop-down menu, choose **Deploy API** and then choose a new or existing stage to deploy the API.

Note the resulting **Invoke URL**. You need it to invoke the API. Before doing that, you must set up the vpcLinkId stage variable.
- i. In the **Stage Editor**, choose the **Stage Variables** tab and choose **Add Stage Variable**.
 - i. Under the **Name** column, type vpcLinkId.
 - ii. Under the **Value** column, type the ID of VPC_LINK, for example, gix6s7.
 - iii. Choose the check-mark icon to save this stage variable.

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

This completes creating the API. You can test invoking the API as with other integrations.

Build an API Gateway API with AWS Integration

Both the [Build an API Gateway API with Lambda Proxy Integration \(p. 19\)](#) and [Build an API Gateway API with Lambda Integration \(p. 18\)](#) 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 [Samples and Tutorials \(p. 507\)](#).

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 API with Lambda Integration \(p. 18\)](#). If you have not yet completed that walkthrough, we suggest that you do it first.

Topics

- [Prerequisites \(p. 76\)](#)
- [Step 1: Create the Resource \(p. 76\)](#)
- [Step 2: Create the GET Method \(p. 77\)](#)
- [Step 3: Create the AWS Service Proxy Execution Role \(p. 77\)](#)
- [Step 4: Specify Method Settings and Test the Method \(p. 78\)](#)
- [Step 5: Deploy the API \(p. 79\)](#)
- [Step 6: Test the API \(p. 79\)](#)
- [Step 7: Clean Up \(p. 80\)](#)

Prerequisites

Before you begin this walkthrough, do the following:

1. Complete the steps in [Get Ready to Build an API Gateway API \(p. 6\)](#).
2. Ensure that the IAM user has access to create policies and roles in IAM. You need to create an IAM policy and role in this walkthrough.
3. Create a new API named `MyDemoAPI`. For more information, see [Build an API with HTTP Custom Integration \(p. 44\)](#).
4. Deploy the API at least once to a stage named `test`. For more information, see [Deploy the API \(p. 36\)](#) in [Build an API Gateway API with Lambda Integration \(p. 18\)](#).
5. Complete the rest of the steps in [Build an API Gateway API with Lambda Integration \(p. 18\)](#).
6. 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 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. If `MyDemoAPI` is displayed, choose **Resources**.
3. In the **Resources** pane, choose the resource root, represented by a single forward slash (/), and then choose **Create Resource**.
4. For **Resource Name**, type `MyDemoAWSProxy`, and then choose **Create Resource**.

Step 2: 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. In the **Resources** pane, choose `/mydemoawsproxy`, and then choose **Create Method**.
2. For the HTTP method, choose **GET**, and then save your choice.

Step 3: Create the AWS Service Proxy Execution Role

In this step, you create an IAM role that your AWS service proxy uses to interact with the AWS service. We call this IAM role an AWS service proxy execution role. Without this role, API Gateway cannot interact with the AWS service. In later steps, you specify this role in the settings for the GET method you just created.

To create the AWS service proxy execution role and its policy

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. Choose **Policies**.
3. Do one of the following:
 - If the **Welcome to Managed Policies** page appears, choose **Get Started**, and then choose **Create Policy**.
 - If a list of policies appears, choose **Create Policy**.
4. Next to **Create Your Own Policy**, choose **Select**.
5. For **Policy Name**, type a name for the policy (for example, `APIGatewayAWSProxyExecPolicy`).
6. For **Description**, type `Enables API Gateway to call AWS services`.
7. For **Policy Document**, type the following, and then choose **Create Policy**.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Resource": ["  
                "*"  
            ],  
            "Action": [  
                "sns>ListTopics"  
            ]  
        }  
    ]  
}
```

This policy document allows the caller to get a list of the Amazon SNS topics for the AWS account.

8. Choose **Roles**.
9. Choose **Create Role**.
10. Choose **AWS Service** under **Select role type** and then choose **API Gateway**.
11. Choose **Next: Permissions**.

12. Choose **Next: Review**.
13. For **Role Name**, type a name for the execution role (for example, `APIGatewayAWSProxyExecRole`), optionally, type a description for this role, and then choose **Create role**.
14. In the **Roles** list, choose the role you just created. You may need to scroll down the list.
15. For the selected role, choose **Attach policy**.
16. Select the check box next to the policy you created earlier (`APIGatewayAWSProxyExecPolicy`) and choose **Attach policy**.
17. The role you just created has the following trust relationship that enables API Gateway assume to role for any actions permitted by the attached policies:

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

For **Role ARN**, note of the Amazon Resource Name (ARN) for the execution role. You need it later. The ARN should look similar to: `arn:aws:iam::123456789012:role/APIGatewayAWSProxyExecRole`, where 123456789012 is your AWS account ID.

Step 4: Specify Method Settings and Test the Method

In this step, you specify the settings for the GET method so that it can interact with an AWS service through an AWS service proxy. You then test the method.

To specify settings for the GET method and then test it

1. In the API Gateway console, in the **Resources** pane for the API named `MyDemoAPI`, in `/mydemoawsproxy`, choose **GET**.
2. In the **Setup** pane, for **Integration type**, choose **Show advanced**, and then choose **AWS Service Proxy**.
3. For **AWS Region**, choose the name of the AWS Region where you want to get the Amazon SNS topics.
4. For **AWS Service**, choose **SNS**.
5. For **HTTP method**, choose **GET**.
6. For **Action**, type **ListTopics**.
7. For **Execution Role**, type the ARN for the execution role.
8. Leave **Path Override** blank.
9. Choose **Save**.
10. In the **Method Execution** pane, in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** displays a 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. In the **Resources** pane, choose **Deploy API**.
2. For **Deployment stage**, choose **test**.
3. For **Deployment description**, type **Calling AWS service proxy walkthrough**.
4. 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 **Stage Editor** pane, next to **Invoke URL**, copy the URL to the clipboard. It should look like this:

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

2. Paste the URL into the address box of a new browser tab.
3. Append /mydemoawsproxy so that it 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 [Create an API as an Amazon S3 Proxy \(p. 526\)](#), [Create an API Gateway API for AWS Lambda Functions \(p. 507\)](#), or [Create an API Gateway API as an Amazon Kinesis Proxy \(p. 551\)](#).

Creating an API in Amazon API Gateway

In Amazon API Gateway, you build an API with 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 \(p. 252\)](#), a [Lambda authorizer \(p. 272\)](#), or an [Amazon Cognito user pool \(p. 286\)](#). 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

- [Choose an Endpoint Type to Set up an API Gateway API \(p. 81\)](#)
- [Initialize API Setup in API Gateway \(p. 82\)](#)
- [Set up API Methods in API Gateway \(p. 106\)](#)
- [Set up API Integrations in API Gateway \(p. 118\)](#)
- [Set up Gateway Responses to Customize Error Responses \(p. 155\)](#)
- [Set up API Gateway Request and Response Data Mappings \(p. 162\)](#)
- [Support Binary Payloads in API Gateway \(p. 199\)](#)
- [Enable Payload Compression for an API \(p. 217\)](#)
- [Enable Request Validation in API Gateway \(p. 221\)](#)
- [Update and Maintain an API in Amazon API Gateway \(p. 233\)](#)
- [Import an API into API Gateway \(p. 238\)](#)

Choose an Endpoint Type to Set up an API Gateway API

An [API endpoint \(p. 3\)](#) refers to a host name of the API. the API endpoint can be edge-optimized or regional, depending on where the majority of your API traffic originates from. You choose a specific endpoint type when creating an API.

An edge-optimized API endpoint optimizes access to an API by geographically distributed clients through an Amazon CloudFront distribution. API requests are routed to the nearest CloudFront Point of Presence (POP). By default, an API is created with the edge-optimized endpoint.

A regional API 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 an edge-optimized API, you create a custom domain name that applies across all the regions. For a regional API, you create a custom domain name that is specific to the API hosting region. It is possible that a regional API deployed in different regions can have the same custom domain name.

Initialize API Setup in API Gateway

For this example, we use a simplified [PetStore](#) API, with the HTTP integration, that exposes the `GET /pets` and `GET /pets/{petId}` methods. The methods are integrated with the two HTTP endpoints, respectively, of `http://petstore-demo-endpoint.execute-api.com/petstore/pets` and `http://petstore-demo-endpoint.execute-api.com/petstore/pets/{petId}`. The API handles `200 OK` responses. The examples focus on the essential programming tasks for creating an API in API Gateway, taking advantage of default settings when possible.

Because of the default settings, the resulting API is edge-optimized. An alternative is to [set up a regional API \(p. 103\)](#). To set up a regional API, you must set explicitly the endpoint type of the API as `REGIONAL`. To set up an edge-optimized API explicitly, you can set `EDGE` as the type of the `endpointConfiguration`.

When setting up an API, you must choose a region. When deployed, the API is region-specific. For an edge-optimized API, the base URL is of the `http[s]://{{restapi-id}}.execute-api.amazonaws.com/stage` format, where `restapi-id` is the API's `id` value generated by API Gateway. 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.

Topics

- [Set up an API Using the API Gateway Console \(p. 82\)](#)
- [Set up an Edge-Optimized API Using AWS CLI Commands \(p. 83\)](#)
- [Set up an Edge-Optimized API Using the AWS SDK for Node.js \(p. 87\)](#)
- [Set up an Edge-Optimized API Using the API Gateway REST API \(p. 94\)](#)
- [Set up an Edge-Optimized API by Importing Swagger Definitions \(p. 101\)](#)
- [Set up a Regional API in API Gateway \(p. 103\)](#)

Set up an API Using the API Gateway Console

To set up an API Gateway API using the API Gateway console, see [Build an API with HTTP Custom Integration \(p. 44\)](#).

You can learn how to set up an API by following an example. For more information, see [Build an API Gateway API from an Example \(p. 9\)](#).

Alternatively, you can set up an API by using the API Gateway [Import API \(p. 238\)](#) feature to upload an external API definition, such as one expressed in the [Swagger 2.0](#) with the [API Gateway Extensions to Swagger \(p. 486\)](#). The example provided in [Build an API Gateway API from an Example \(p. 9\)](#) 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:

```
{  
  "name": "Simple PetStore (AWS CLI)",  
  "id": "vaz7da96z6",  
  "createdDate": 1494572809  
}
```

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": [  
    {  
      "path": "/",  
      "id": "begaltmsm8"  
    }  
  ]  
}
```

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:

```
{  
  "path": "/pets",  
  "pathPart": "pets",  
  "id": "6sxz2j",
```

```
        "parentId": "begaltmsm8"  
    }
```

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:

```
{  
    "path": "/pets/{petId}",  
    "pathPart": "{petId}",  
    "id": "rjkmth",  
    "parentId": "6sxz2j"  
}
```

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:

```
{  
    "apiKeyRequired": false,  
    "httpMethod": "GET",  
    "authorizationType": "NONE"  
}
```

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 [Controlling Access to an API \(p. 245\)](#).

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:

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "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:

```
{
    "httpMethod": "GET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "HTTP",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
    "cacheNamespace": "6sxz2j"
}
```

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:

```
{
  "passthroughBehavior": "WHEN_NO_MATCH",
  "cacheKeyParameters": [],
  "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
  "httpMethod": "GET",
  "cacheNamespace": "rjkmth",
  "type": "HTTP",
  "requestParameters": {
    "integration.request.path.id": "method.request.path.petId"
  }
}
```

- 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:

```
{
  "selectionPattern": "",
  "statusCode": "200"
}
```

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'
```

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://petstore-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://petstore-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`:

```
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 Using the API Gateway REST API

Setting up an API using the API Gateway REST API involves working with API Gateway resources of the [RestApi](#), [Resource](#), [Method](#), [MethodResponse](#), [Integration](#), and [IntegrationResponse](#) types. The following procedure walks through the basic steps to work with these API Gateway resources to set up the simple PetStore API.

To create the simple PetStore API

1. To set up an edge-optimized API, invoke the API Gateway's [restapi:create](#) link-relation to add an API Gateway resource of [RestApi](#) to your account in a chosen region:

```
POST /restapis HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c
Cache-Control: no-cache
Postman-Token: 0889d2b5-e507-6aab-f222-ab9548dbacaa

{
  "name": "Simple PetStore (REST API)",
  "description": "A sample API Gateway API created using the REST API."
}
```

The successful request returns a response of the 201 Created status code with a payload similar to the following output:

```
{
  "createdDate": "2017-05-11T21:47:24Z",
  "description": "A sample API Gateway API created using the REST API.",
  "endpointConfiguration": {
    "types": "EDGE"
  },
  "id": "x7hyqq0ik7",
  "name": "Simple PetStore (REST API)"
}
```

Note of the `id` value of the newly created `RestApi`. You will use this `id` value in subsequent operations on this API. A newly created `RestApi` comes with the API's root resource (/) of the API.

You need to specify the `id` value of this root resource to append a child resource, and to add a method on the root resource. To get this API identifier, get the API's `resources` collection and then parse the result to obtain the `id` property value of the entry with the path value of `/`.

2. To get the API root resource identifier, invoke the `restapi:resources` link-relation:

```
GET /restapis/x7hyqq0ik7/resources HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T215738Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-
west-2/apigateway/aws4_request, SignedHeaders=content-type;host;x-amz-date,
Signature=76c24ef91d835b85313142bf75545c4ac4c212067e8188ee6a127c21dae09e29
```

The request returns a response of the `200 OK` status code with a payload similar to the following output:

```
{
  "_embedded": {
    "item": {
      "_links": {
        "self": {
          "href": "/restapis/x7hyqq0ik7/resources/0f72nvvnkd"
        },
        "method:by-http-method": {
          "href": "/restapis/x7hyqq0ik7/resources/0f72nvvnkd/methods/{http_method}",
          "templated": true
        },
        "method:put": {
          "href": "/restapis/x7hyqq0ik7/resources/0f72nvvnkd/methods/{http_method}",
          "templated": true
        },
        "resource:create-child": {
          "href": "/restapis/x7hyqq0ik7/resources/0f72nvvnkd"
        },
        "resource:update": {
          "href": "/restapis/x7hyqq0ik7/resources/0f72nvvnkd"
        }
      },
      "id": "0f72nvvnkd",
      "path": "/"
    }
  }
}
```

The root resource identifier is the `id` value associated with the path value of `" / "`. In this example, it is `0f72nvvnkd`.

3. To add a `pets` resource under the root resource (`0f72nvvnkd`) to represent the `pets` collection of the pet store, call the `resource:create` link-relation of API Gateway.

```
POST /restapis/x7hyqq0ik7/resources/0f72nvvnkd HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T000729Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=8a7093411c97b0aa90f4b1890475d93cf20aa3732089da61e6deb410fbc6037d
Cache-Control: no-cache
Postman-Token: 48abcd2f-c357-9e44-669e-d8d813f876ca

{
  "pathPart": "pets"
```

```
}
```

The successful response contains the newly created child resource (`47rxl6`, `pets` or `/pets`) and its parent (`0f72nvvnkd`).

```
{
  ...
  "id": "47rxl6",
  "parentId": "0f72nvvnkd",
  "path": "/pets",
  "pathPart": "pets"
}
```

Similarly, to add an individual pet under the `pets` collection (as referenced the resource Id of `47rxl6`, invoke the following `resource:create` link-relation:

```
POST /restapis/x7hyqq0ik7/resources/47rxl6 HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T000729Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=8a7093411c97b0aa90f4b1890475d93cf20aa3732089da61e6deb410fbc6037d
Cache-Control: no-cache
Postman-Token: 48abcd2f-c357-9e44-669e-d8d813f876ca

{
  "pathPart": "{petId}"
}
```

The resulting response looks like this one:

```
{
  ...
  "id": "ab34fgd",
  "parentId": "47rxl6",
  "path": "/pets/{petId}",
  "pathPart": "{petId}"
}
```

4. To add a `GET` method to the API's `/pets` resource (`47rxl6`), invoke the following `method:put` link-relation of API Gateway:

```
PUT /restapis/x7hyqq0ik7/resources/47rxl6/methods/GET HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T000729Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=8a7093411c97b0aa90f4b1890475d93cf20aa3732089da61e6deb410fbc6037d
Cache-Control: no-cache
Postman-Token: 48abcd2f-c357-9e44-669e-d8d813f876ca

{
  "authorizationType": "NONE"
}
```

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 [Controlling Access to an API \(p. 245\)](#).

The successful request returns a 201 Created response with a payload similar to the following:

```
{  
    ...  
    "apiKeyRequired": false,  
    "authorizationType": "NONE",  
    "httpMethod": "GET"  
}
```

To add a GET method to the API's /pets/{petId} resource (ab34fgd), invoke the following [method:put](#) link-relation of API Gateway:

```
PUT /restapis/x7hyqq0ik7/resources/ab34fgd/methods/GET HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170512T000729Z  
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature=8a7093411c97b0aa90f4b1890475d93cf20aa3732089da61e6deb410fbc6037d  
Cache-Control: no-cache  
Postman-Token: 48abcd2f-c357-9e44-669e-d8d813f876ca  
  
{  
    "authorizationType": "NONE",  
    "requestParameters": {  
        "method.request.path.petId": true  
    }  
}
```

You must declare the method request path parameter of `petId` for API Gateway to map its dynamically set value to the corresponding integration request parameter before passing it to the backend. You must always set a path parameter as required. In addition, depending on API requirements, you can set up header and query parameters on a method request. For POST, PUT, PATCH, or any other method taking a payload, you can define a model for the payload in the method request. For more information about these settings, see [Set up API Methods in API Gateway \(p. 106\)](#).

The successful response has a status code of 201 Created and a payload similar to the following:

```
{  
    "apiKeyRequired": false,  
    "authorizationType": "NONE",  
    "httpMethod": "GET",  
    "requestParameters": {  
        "method.request.path.petId": true  
    }  
}
```

5. To add a method response of the 200 status code for the GET /pets method of the API, invoke the [methodresponse:put](#) link-relation, specifying 47rx16 to reference the resource exposing this method:

```
PUT /restapis/x7hyqq0ik7/resources/47rx16/methods/GET/responses/200 HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170512T003943Z
```

```
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=229ef4cfba4bbe41132f36c027f0ae4449bb741671875075a4b216e9b778233e
Cache-Control: no-cache
Postman-Token: 268fcf18-92e4-dfea-821a-ebf4e1d0edfd
{}
```

The successful request returns a response of the 201 Created status code with a payload similar to the following output:

```
{
  ...
  "statusCode": "200"
}
```

Similarly, to add a 200 response to the GET /pets/{petId} method, invoke the following methodResponse:put link-relation, referencing the desired /pets/{petId} resource by its ID (ab34fgd):

```
PUT /restapis/x7hyqq0ik7/resources/ab34fgd/methods/GET/responses/200 HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T003943Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=229ef4cfba4bbe41132f36c027f0ae4449bb741671875075a4b216e9b778233e
Cache-Control: no-cache
Postman-Token: 268fcf18-92e4-dfea-821a-ebf4e1d0edfd
{}
```

The API Gateway resources of Method and MethodResponse that you just set up define the client-facing interface of the API. For non-proxy integrations, you must add and configure an API Gateway resource of Integration and IntegrationResponse to encapsulate the integration request submitted to the backend and the integration response returned by the backend. For proxy integrations, however, you do not set up Integration and IntegrationResponse.

6. To set up Integration for the GET /pets method, invoke the [integration:put](#) link-relation of API Gateway, referencing the /pets resource by its ID value (47rxl6):

```
PUT /restapis/x7hyqq0ik7/resources/47rxl6/methods/GET/integration HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T002249Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=10359971a8c54862a47e39d6a6e4b6e62c263e9a2b785b47b40c426c0aa61c19

{
  "type" : "HTTP",
  "httpMethod" : "GET",
  "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets"
}
```

In the request payload, uri points to the backend endpoint associated with the method. The type refers to the integration type. For the specified HTTP endpoint <http://petstore-demo-endpoint.execute-api.com/petstore/pets>, the integration type must be HTTP. The httpMethod property refers to the HTTP verb as required by the backend, which may be different from the method request HTTP verb set when calling the method:put link-relation.

The successful request returns a response of a 201 Created status code with a payload similar to the following output:

```
{  
  ...  
  "cacheKeyParameters": [],  
  "cacheNamespace": "47rxl6",  
  "httpMethod": "GET",  
  "passthroughBehavior": "WHEN_NO_MATCH",  
  "type": "HTTP",  
  "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets"  
}
```

Similarly, to set up the integration for the GET /pets/{petId} method, invoke the following [integration:put](#) link-relation of API Gateway, referencing the /pets/{petId} resource by its value of ab34fgd, and adding the request parameter mapping from {petId} to {id}:

```
PUT /restapis/x7hyqq0ik7/resources/ab34fgd/methods/GET/integration HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170512T002249Z  
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature=10359971a8c54862a47e39d6a6e4b6e62c263e9a2b785b47b40c426c0aa61c19  
  
{  
  "type" : "HTTP",  
  "httpMethod" : "GET",  
  "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",  
  "requestParameters": {  
    "integration.request.path.id": "method.request.path.petId"  
  }  
}
```

The successful response of this [integration:put](#) request is shown as follows:

```
{  
  ...  
  "cacheKeyParameters": [],  
  "cacheNamespace": "ab34fgd",  
  "httpMethod": "GET",  
  "passthroughBehavior": "WHEN_NO_MATCH",  
  "type": "HTTP",  
  "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",  
  "requestParameters": {  
    "integration.request.path.id": "method.request.path.petId"  
  }  
}
```

7. To set up the 200 OK IntegrationResponse for the GET /pets method, invoke the following [integrationresponse:put](#) link-relation of API Gateway:

```
PUT /restapis/x7hyqq0ik7/resources/47rxl6/methods/GET/integration/responses/200  
HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170512T004542Z  
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature=545ab6ea151f72c52161ee856ee621f136d717e40a02743cd8fe3638895794b1
```

```
Cache-Control: no-cache
Postman-Token: d29cabea-5232-7021-88ad-f1e950f55a99

{}
```

The successful request returns a response of the 201 Created status code and a payload similar to the following output:

```
{
  ...
  "statusCode": "200"
}
```

To set up the 200 OK IntegrationResponse for the GET /pets/{petId} (the resource id is ab34fgd) method, invoke the following [integrationresponse:put](#) link-relation of API Gateway:

```
PUT /restapis/x7hyqq0ik7/resources/ab34fgd/methods/GET/integration/responses/200
HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T004542Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=545ab6ea151f72c52161ee856ee621f136d717e40a02743cd8fe3638895794b1

{}
```

We now have successfully created a simple PetStore API with the GET /pets and GET /pets/{petId} method with the HTTP integration type.

8. To open the API for your customers to call, deploy the API to a test stage by invoking [deployment:create](#) link-relation of API Gateway.

```
POST /restapis/x7hyqq0ik7/deployments HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170512T004542Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170512/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=545ab6ea151f72c52161ee856ee621f136d717e40a02743cd8fe3638895794b1

{
  "stageName" : "test",
  "stageDescription" : "First stage",
  "description" : "First deployment"
}
```

The successful response has a status code of 201 Created and a payload similar to the following:

```
{
  ...
  "createdDate": "2017-10-13T20:28:56Z",
  "description": "First deployment",
  "id": "s7ja1r"
}
```

Now, you can test the deployed API by typing the <https://x7hyqq0ik7.execute-api.us-west-2.amazonaws.com/test/pets> URL, for the GET /pets method, in a browser. Substitute the

RestApi identifier (`x7hyqq0ik7`) 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 the `https://x7hyqq0ik7.execute-api.us-west-2.amazonaws.com/test/pets/3` URL in the browser, replacing the RestApi identifier with the identifier of your API. The expected output should be like this:

```
{  
  "id": 3,  
  "type": "fish",  
  "price": 0.99  
}
```

Set up an Edge-Optimized API by Importing Swagger Definitions

You can set up an API in API Gateway by specifying Swagger definitions of appropriate API Gateway API entities and importing the Swagger definitions into API Gateway.

The following Swagger 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.

```
{  
  "swagger": "2.0",  
  "info": {  
    "title": "Simple PetStore (Swagger)"  
  },  
  "schemes": [  
    "https"  
  ],  
  "paths": {  
    "/pets": {  
      "get": {  
        "responses": {  
          "200": {  
            "description": "200 response"  
          }  
        }  
      }  
    },  
    "x-amazon-apigateway-integration": {  
      "httpMethod": "GET",  
      "uri": "https://petstore.swagger.io/v2/pets",  
      "responses": {  
        "200": {  
          "description": "A pet object that was successfully found"  
        }  
      }  
    }  
  }  
}
```

```
"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 Swagger definitions into API Gateway using the API Gateway console.

To import the simple API Swagger definitions using the API Gateway console

1. Sign in to the API Gateway console.
2. Choose **Create API**.
3. Choose **Import from Swagger**.
4. If you saved the preceding Swagger definitions in a file, choose **Select Swagger File**. You can also copy the Swagger definitions and paste them into the import text editor.
5. Choose **Import** to finish importing the Swagger definitions.

To import the Swagger definitions using the API Gateway REST API, call the [restapi:import](#) action, supplying the preceding Swagger definitions as the payload. For more information, see the example in the Remarks section of the [restapi:import](#) topic.

To import the Swagger definitions using the AWS CLI, save the Swagger definitions into a file and then run the following command, assuming that you use the `us-west-2` region and the absolute Swagger file path is `file:///path/to/API_Swagger_template.json`:

```
aws apigateway import-rest-api --body 'file:///path/to/API_Swagger_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. In addition, for customers to manage their own Amazon CloudFront distribution, they can use a regional API endpoint to ensure that API Gateway does not associate the API with the service-controlled CloudFront distributions.

To create a regional API, you follow the steps in [creating an edge-optimized API \(p. 82\)](#), 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, the AWS SDK for Javascript for Node.js, and the API Gateway REST API.

Topics

- [Create a Regional API Using the API Gateway Console \(p. 103\)](#)
- [Create a Regional API Using the AWS CLI \(p. 103\)](#)
- [Create a Regional API Using the AWS SDK for JavaScript \(p. 104\)](#)
- [Create a Regional API Using the API Gateway REST API \(p. 104\)](#)
- [Test a Regional API \(p. 105\)](#)

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 and choose **+ Create API**.
2. Under **Create new API**, choose the **New API** option.
3. Type a name (for example, `Simple PetStore (Console, Regional)`) for **API name**.
4. Choose **Regional** for **Endpoint Type**.
5. Choose **Create API**.

From here on, you can proceed to set up API methods and their associated integrations as described in [creating an edge optimized API \(p. 44\)](#).

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"] }'
```

The successful response returns an output 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” \(p. 83\)](#) 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);  
    }  
});
```

The successful response returns an output 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 ” \(p. 87\)](#) to set up methods and integrations for this API.

Create a Regional API Using the API Gateway REST API

To create a regional API using the API Gateway REST API, submit a POST request as follows:

```
POST /restapis HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170511T214723Z
```

```
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c

{
    "name": "Simple PetStore (REST API, Regional)",
    "description": "A sample API Gateway API created using the REST API.",
    "endpointConfiguration" : {
        "types" : ["REGIONAL"]
    }
}
```

The successful response has the status code of 201 Created and a body similar to the following output:

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "A sample API Gateway API created using the REST API.",
    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (REST API, Regional)"
}
```

After completing the preceding steps, you can follow the instructions in the section called ["Set up an Edge-Optimized API Using the API Gateway REST API" \(p. 94\)](#) to set up methods and integrations for this API.

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 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 \(p. 329\)](#) and [provide proper error messages \(p. 155\)](#) for [invalid requests \(p. 221\)](#).

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 \(p. 9\)](#), 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 of 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 an identical transformation that passes through the integration as-is. For example, the following `200` method response shows an example of pass-through 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 a 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.

Topics

- [Set up a Method Request in API Gateway \(p. 107\)](#)
- [Set up Method Responses in API Gateway \(p. 113\)](#)
- [Set up a Method Using the API Gateway Console \(p. 115\)](#)

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 \(p. 115\)](#)
- [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 API Setup in API Gateway \(p. 82\)](#).

Topics

- [Set up API Resources \(p. 107\)](#)
- [Set up an HTTP Method \(p. 110\)](#)
- [Set up Method Request Parameters \(p. 110\)](#)
- [Set up Method Request Model \(p. 111\)](#)
- [Set up Method Request Authorization \(p. 112\)](#)
- [Set up Method Request Validation \(p. 112\)](#)

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 \(p. 432\)](#).

For the [PetStore sample API \(p. 9\)](#), 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 following:

```
{
  "items": [
    {
      "path": "/pets",
      "resourceMethods": {
        "GET": {}
      },
      "id": "6sxzz2j",
      "pathPart": "pets",
      "parentId": "svzr2028x8"
    },
    {
      "path": "/pets/{petId}",
      "resourceMethods": {
        "GET": {}
      },
      "id": "rjkmth",
      "pathPart": "{petId}",
      "parentId": "6sxzz2j"
    },
    {
      "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 \(p. 122\)](#) 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 function-based Lambda authorizer, set this property to `CUSTOM`. To leverage 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](#)” (p. 112).

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 “Set up API Integrations” \(p. 118\)](#).

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 \(p. 221\)](#), [generate an SDK \(p. 416\)](#), and initialize a mapping template for setting up the integration in the API Gateway console. For information about how to create a [model](#), see [Models and Mapping Templates \(p. 164\)](#).

Depending on the content types, a method payload can have different formats. A model is indexed against the media type of the applied payload. 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.

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 function-based Lambda authorizer (`CUSTOM`). The API Gateway console sets `NONE` for open access as the default.

To use IAM permissions to authorize access to the API method, set the `authorization-type` input property to `AWS_IAM`. When this option is set, API Gateway verifies the caller's signature on the request, based on the caller's IAM user's access key identifier and secret key. If the verified user has permission to call the method, the request is accepted. Otherwise, the request is rejected and the caller receives an unauthorized error response. The call to the method does not succeed unless the caller has been granted permission to invoke the API method or if the caller is allowed to assume a role that has been granted the permission. The caller has permissions to call this and any other API methods created by anyone of the same AWS account if the caller has the following IAM policy attached to his or her IAM user:

```
{  
    "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” \(p. 252\)](#).

At present, such a policy can be granted to only the IAM users of the API owner's account. Users from a different AWS account can call the API methods if they are allowed to assume a role of the API owner account and the assumed role has the proper permissions for 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” \(p. 272\)](#).

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 Pools” \(p. 286\)](#).

Set up Method Request Validation

You can enable request validation when setting up an API method request. You need to first to 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": true,
    "validateRequestBody": true,
    "id": "jgpwy6",
    "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 jgpwy6
```

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 prescribed 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 \(p. 134\)](#) 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, the `petId` path variable, and the `query` query parameter as the mapping targets:

```
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.request.header.my-
header=false,method.request.path.petId=true,method.request.querystring.query=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 \
    --request-parameters method.request.header.my-
header=false,method.request.path.petId=true,method.request.querystring.query=false
    --request-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

Before setting up an API method, verify the following:

- You must have the method available in API Gateway. Follow the instructions in [Build an API with HTTP Custom Integration \(p. 44\)](#).
- 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 the Lambda Custom Integration \(p. 29\)](#) of the [Build an API Gateway API with Lambda Integration \(p. 18\)](#).
- 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 \(p. 298\)](#).

Topics

- [Set up an API Gateway Method Request in the API Gateway Console \(p. 115\)](#)
- [Set up an API Gateway Method Response Using the API Gateway Console \(p. 117\)](#)

Set up an API Gateway Method Request in the API Gateway Console

To use the API Gateway console to specify an API's method request/response, and to configure how the method will authorize requests, 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 \(p. 123\)](#). They are best used to supplement the discussions given in [Build an API Gateway API with Lambda Integration \(p. 18\)](#).

1. With the method selected in the **Resources** pane, choose **Method Request** from the **Method Execution** pane.

2. Under **Settings**, choose the pencil icon to open the **Authorization** drop-down menu and choose one of the available authorizers.
 - a. To enable open access to the method for any user, choose **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, choose **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 value of the **ARN** field in the **Authorization Settings** section. For more information about setting the IAM permissions, see [Control Access to an API with IAM Permissions \(p. 252\)](#).

To create the IAM role, you can adapt the instructions in "To create the Lambda invocation role and its policy" and "To create the Lambda execution role and its policy" in the [Create Lambda Functions \(p. 29\)](#) section of the [Build an API Gateway API with Lambda Integration \(p. 18\)](#).

- To save your choice, choose **Update**. Otherwise, choose **Cancel**.
- c. To use a Lambda authorizer, choose one under **Token authorizer**. You must have created a Lambda authorizer to have this choice displayed in the drop-down menu. For information on how to create a Lambda authorizer, see [Use API Gateway Lambda Authorizers \(p. 272\)](#).
 - d. To use an Amazon Cognito user pool, choose an available user pool under **Cognito user pool authorizers**. You must have created a user pool in Amazon Cognito and an Amazon Cognito user pool authorizer in API Gateway to have this choice displayed in the drop-down menu. For information on how to create an Amazon Cognito user pool authorizer, see [Use Amazon Cognito User Pools \(p. 286\)](#).
 3. To enable or disable request validation, choose the pencil icon from the **Request Validator** drop-down menu and choose one of the listed options. For more information about each option, see [Enable Request Validation in API Gateway \(p. 221\)](#).
 4. To require an API key, choose the pencil icon to open the **API Key Required** drop-down menu and choose either **true** or **false** according to your API requirements. When enabled, API keys are used in [usage plans \(p. 314\)](#) to throttle client traffic.
 5. To add a query string parameter to the method, do the following:
 - a. Choose the arrow next to **URL Query String Parameters**, and then choose **Add query string**.
 - b. For **Name**, type the name of the query string parameter.
 - c. Choose the check-mark icon to save the new query string parameter name.
 - d. If the newly created query string parameter is to be used for request validation, choose the **Required** option. For more information about the request validation, see [Enable Request Validation in API Gateway \(p. 221\)](#).

- e. If the newly created query string parameter is to be used as part of a caching key, check the **Caching** option. This is applicable only when caching is enabled. For more information about caching, see [Use Method/Integration Parameters as Cache Keys \(p. 379\)](#).

Tip

To remove the query string parameter, choose the x icon associated with it and then choose **Remove this parameter and any dependent parameters** to confirm the removal.

To change the name of the query string parameter, remove it and then create a new one.

6. To add a header parameter to the method, do the following:
 - a. Choose the arrow next to **HTTP Request Headers**, and then choose **Add header**.
 - b. For **Name**, type the name of the header parameter and then choose the check-mark icon to save the settings.
 - c. If the newly created header parameter is to be used for request validation, choose the **Required** option. For more information about request validation, see [Enable Request Validation in API Gateway \(p. 221\)](#).
 - d. If the newly created header parameter is to be used as part of a caching key, choose the **Caching** option. This is applicable only when caching is enabled. For more information about caching, see [Use Method/Integration Parameters as Cache Keys \(p. 379\)](#).

Tip

To remove the header parameter, choose the x icon associated with it and then choose **Remove this parameter and any dependent parameters** to confirm the removal.

To change the name of the header parameter, remove it and then create a new one.

7. To declare the payload format of a method request with the **POST**, **PUT**, or **PATCH** HTTP verb, expand **Request Body**, and do the following:
 - a. Choose **Add model**.
 - b. Type a MIME-type (for example, `application/json`) for **Content type**.
 - c. Open the **Model name** drop-down menu to choose an available model for the payload and choose the check-mark icon to save the settings.

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 [Create a Model \(p. 170\)](#).

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.

8. To assign an operation name in a Java SDK of this API, generated by API Gateway, expand **SDK Settings** and type a name in **Operation 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`.

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. Choose **Method Response** from **Method Execution** for a given method of an API resource.
2. To add a new response, choose **Add Response**.
 - a. Type an HTTP status code; for example, 200, 400, or 500) for **HTTP Status**, and then choose the check-mark icon to save the choice.

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.
 - b. Expand the response of the given status code.
 - c. Choose **Add Header**.
 - d. Type a name for **Name** under **Response Headers for {status}**, and then choose the check-mark icon to save the choice.

If you need to translate any backend-returned header to one defined in a method response, you must first add the method response header as described in this step .
 - e. Choose **Add Response Model** under **Response Body for {status}**.
 - f. Type the media type of the response payload for **Content type** and choose a model from the **Models** drop-down menu.
 - g. Choose the check-mark icon to save the settings.
3. To modify an existing response, expand the response and follow Step 2 above.
4. To remove a response, choose the **x** icon for the response and confirm you want to delete the 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.

Set up API Integrations in API Gateway

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 may or may 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; 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 \(p. 119\)](#)
- [Set up an Integration Response in API Gateway \(p. 125\)](#)
- [Set up Lambda Integrations in API Gateway \(p. 126\)](#)
- [Set up HTTP Integrations in API Gateway \(p. 142\)](#)
- [Set up API Gateway Private Integrations \(p. 146\)](#)
- [Set up Mock Integrations in API Gateway \(p. 152\)](#)

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 \(p. 119\)](#)
- [Choose an API Gateway API Integration Type \(p. 121\)](#)
- [Set up a Proxy Integration with a Proxy Resource \(p. 122\)](#)
- [Set up an API Integration Request Using the API Gateway Console \(p. 123\)](#)

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 \(p. 162\)](#).

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 \(p. 162\)](#), 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. It is intended for calling all AWS service actions, but is not recommended for calling a Lambda function, because the Lambda custom

integration is a legacy technology. With the 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 \(p. 132\)](#) 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 \(p. 134\)](#). 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 \(p. 155\)](#) are other examples of mock integrations.

Set up a Proxy Integration with a Proxy Resource

To set up a proxy resource in an API Gateway API with a proxy integration, you perform the following three 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 [Known Issues \(p. 585\)](#).

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 the 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](#) (p. 585).

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/a0/a1/.../aN`, `/site/b0/b1/.../bM`, 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 `[a0, a1, ..., aN, b0, b1, ...bM, ...]`. 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 procedure describes how to use the API Gateway console to specify method settings.

1. In the **Resources** pane, choose the method.
2. In the **Method Execution** pane, choose **Integration Request**. For **Integration type**, choose one of the following:
 - Choose **Lambda Function** if your API will be integrated with a Lambda function. At the API level, this is an AWS integration type.
 - Choose **HTTP** if your API will be integrated with an HTTP endpoint. At the API level, this is the HTTP integration type.
 - Choose **AWS Service** if your API will be integrated directly with an AWS service. At the API level, this is the AWS integration type. The **Lambda Function** option above is a special case of the AWS integration for invoking a Lambda function and is available only in the API Gateway console. To set up an API Gateway API 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, you must choose the **AWS Service** option here.
 - Choose **Mock** 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. If you choose this option, skip the rest of the instructions in this topic and see [Set up Mock Integrations in API Gateway \(p. 152\)](#).

3. If you chose **Lambda Function**, do the following:
 - a. For **Lambda Region**, choose the region identifier that corresponds to the region where you created the Lambda function. For example, if you created the Lambda function in the US East (N. Virginia) Region, choose **us-east-1**. For a list of region names and identifiers, see [AWS Lambda](#) in the *Amazon Web Services General Reference*.
 - b. For **Lambda Function**, type the name of the Lambda function, and then choose the function's corresponding ARN.
 - c. Choose **Save**.
4. If you chose **HTTP**, do the following:
 - a. For **HTTP method**, choose the HTTP method type that most closely matches the method in the HTTP backend.
 - b. For **Endpoint URL**, type the URL of the HTTP backend you want this method to use.
 - c. Choose **Save**.
5. If you chose **Mock**, do the following:
 - Choose **Save**.
6. If you chose **AWS Service**, do the following:
 - a. For **AWS Region**, choose the AWS Region you want this method to use to call the action.
 - b. For **AWS Service**, choose the AWS service you want this method to call.
 - c. For **AWS Subdomain**, type 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.
 - d. 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**.
 - e. For **Action**, type the action you want to use. For a list of available actions, see the API reference documentation for the AWS service you chose for **AWS Service**.
 - f. For **Execution Role**, type 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 "To create the Lambda invocation role and its policies" and "To create the Lambda execution role and its policy" in the [Create Lambda Functions \(p. 29\)](#) section. 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"  
        }  
    ]  
}
```

- g. If the action you typed for **Action** provides a custom resource path that you want this method to use, for **Path Override**, type this custom resource path. For the custom resource path, see the API reference documentation for the AWS service you chose for **AWS Service**.
- h. Choose **Save**.

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 \(p. 138\)](#). 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 [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

Set up Lambda Integrations in API Gateway

You can integrate an API method with a Lambda function using Lambda proxy integration or Lambda custom integration.

With the proxy integration, the setup is simple. If your API does not require content encoding or caching, you only need to 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 the credential to an IAM role with permissions to allow API Gateway to call the Lambda function on your behalf.

With the custom integration, the setup is more involved. 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 \(p. 126\)](#)
- [Set up Lambda Custom Integrations in API Gateway \(p. 134\)](#)
- [Handle Lambda Errors in API Gateway \(p. 138\)](#)

Set up Lambda Proxy Integrations in API Gateway

Topics

- [Understand API Gateway Lambda Proxy Integration \(p. 126\)](#)
- [Set up a Proxy Resource with the Lambda Proxy Integration \(p. 128\)](#)
- [Set up Lambda Proxy Integration Using the AWS CLI \(p. 129\)](#)
- [Input Format of a Lambda Function for Proxy Integration \(p. 132\)](#)
- [Output Format of a Lambda Function for Proxy Integration \(p. 134\)](#)

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.

With the Lambda proxy integration, when a client submits an API request, API Gateway passes to the integrated Lambda function the raw request as-is. This [request data \(p. 132\)](#) 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 \(p. 134\)](#).

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 the API Gateway proxy integrations, see [Set up a Proxy Integration with a Proxy Resource \(p. 122\)](#).

The 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.

Set up a Proxy Resource with the 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 a Swagger 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 Lambda proxy integration, see [Build an API Gateway API with Lambda Proxy Integration \(p. 19\)](#).

The following Swagger API definition file shows an example of an API with a proxy resource that is integrated with the [SimpleLambda4ProxyResource \(p. 20\)](#) Lambda function.

```
{
  "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"
          }
        ]
      }
    }
  }
}
```

```

        "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"
}
}
}
}
```

With the 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 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 \(p. 132\)](#). 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 \(p. 134\)](#).

With the 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.

As an example, we use the following sample Lambda function as the backend of the API:

```

exports.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.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 \(p. 135\)](#), 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 vaz7da96z6 \
    --resource-id 2jf6xt \
    --http-method ANY \
    --type AWS_PROXY \
    --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 \
    --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole

```

For Lambda integrations, you must use the HTTP method of `POST` for the integration request. The IAM role of `apigAwsProxyRole` must have policies allowing the `apigateway` service to invoke Lambda functions. For more information about the IAM permissions, see [the section called "API Gateway Permissions Model for Creating and Managing an API" \(p. 252\)](#).

The 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-1:1234567890: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
```

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' \
-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'
```

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 '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' \
-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 \
-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' \
-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

With the Lambda proxy integration, API Gateway maps the entire client request to the `input` event parameter of the backend Lambda function as follows:

```
{
  "resource": "Resource path",
  "path": "Path parameter",
  "httpMethod": "Incoming request's method name"
  "headers": {Incoming request headers}
  "queryStringParameters": {query string parameters }
  "pathParameters": {path parameters}
  "stageVariables": {Applicable stage variables}
  "requestContext": {Request context, including authorizer-returned key-value pairs}
  "body": "A JSON string of the request payload."
```

```

    "isBase64Encoded": "A boolean flag to indicate if the applicable request payload is
    Base64-encode"
}

```

We illustrate this using the following POST request to show an API deployed to testStage with a stage variable of stageVariableName=stageVariableValue:

```

POST /testStage/hello/world?name=me HTTP/1.1
Host: gy415nuibc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
headerName: headerValue

{
    "a": 1
}

```

This request produces the following response payload, which contains the output returned from the backend Lambda function, where input was set to the event parameter to the Lambda function.

```

{
    "message": "Hello me!",
    "input": {
        "resource": "/{proxy+}",
        "path": "/hello/world",
        "httpMethod": "POST",
        "headers": {
            "Accept": "*/*",
            "Accept-Encoding": "gzip, deflate",
            "cache-control": "no-cache",
            "CloudFront-Forwarded-Proto": "https",
            "CloudFront-Is-Desktop-Viewer": "true",
            "CloudFront-Is-Mobile-Viewer": "false",
            "CloudFront-Is-SmartTV-Viewer": "false",
            "CloudFront-Is-Tablet-Viewer": "false",
            "CloudFront-Viewer-Country": "US",
            "Content-Type": "application/json",
            "headerName": "headerValue",
            "Host": "gy415nuibc.execute-api.us-east-1.amazonaws.com",
            "Postman-Token": "9f583ef0-ed83-4a38-aef3-eb9ce3f7a57f",
            "User-Agent": "PostmanRuntime/2.4.5",
            "Via": "1.1 d98420743a69852491bbdea73f7680bd.cloudfront.net (CloudFront)",
            "X-Amz-Cf-Id": "pn-PWIJc6thYnZm5P0NMgOUGlL1DYtl0gdeJky8tqsg8iS_sgsKD1A==",
            "X-Forwarded-For": "54.240.196.186, 54.182.214.83",
            "X-Forwarded-Port": "443",
            "X-Forwarded-Proto": "https"
        },
        "queryStringParameters": {
            "name": "me"
        },
        "pathParameters": {
            "proxy": "hello/world"
        },
        "stageVariables": {
            "stageVariableName": "stageVariableValue"
        },
        "requestContext": {
            "accountId": "12345678912",
            "resourceId": "roq9wj",
            "stage": "testStage",
            "requestId": "deef4878-7910-11e6-8f14-25afc3e9ae33",
            "identity": {
                "cognitoIdentityPoolId": null,
                "accountId": null,

```

```

        "cognitoIdentityId": null,
        "caller": null,
        "apiKey": null,
        "sourceIp": "192.168.196.186",
        "cognitoAuthenticationType": null,
        "cognitoAuthenticationProvider": null,
        "userArn": null,
        "userAgent": "PostmanRuntime/2.4.5",
        "user": null
    },
    "resourcePath": "/{proxy+}",
    "httpMethod": "POST",
    "apiId": "gy415nuibc"
},
"body": "{\r\n\t\"a\": 1\r\n}",
"isBase64Encoded": false
}
}

```

In the input to Lambda, the `requestContext` object is a map of key-value pairs. The key is a property name of the [\\$context \(p. 191\)](#) variable and the value is the property value of the corresponding `$context` variable. API Gateway may add new keys to the map. Depending on the features enabled, the `requestContext` map may vary from API to API. For example, in the preceding example, `$context.authorizer.*` properties are absent because no Lambda authorizer (formerly known as a custom authorizer) is enabled for the API.

Note

API Gateway enacts certain restrictions and limitations when handling methods with either Lambda proxy integration or HTTP proxy integration. For details, see [Known Issues \(p. 585\)](#).

Output Format of a Lambda Function for Proxy Integration

With the 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", ... },
    "body": ...
}
```

In the output, `headers` can be unspecified if no extra response headers are to be returned. 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 and set `isBase64Encoded` to `true`. Otherwise, you can set it to `false` or leave it unspecified.

If the function output is of a different format, API Gateway returns a `502 Bad Gateway` error response.

In a Lambda function in Node.js, to return a successful response, 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.

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. 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`.

For reference, a Node.js version of the Lambda function is shown as follows:

```
exports.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);
    }
};
```

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 vaz7da96z6 \
    --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 file://path/to/integration-request-template.json \
    --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole
```

Here, the `request-template` parameter value, `file://path/to/integration-request-template.json`, points to a JSON file, named `integration-request-template.json` in the `path/to` directory, which contains a key-value map as a JSON object. The key is a media type of the request payload and the value is a mapping template for the body of the specified content type. In this example, the JSON file contains the following JSON object:

```
{"application/json": "{\"greeter\": \"$input.params('greeter')\"}"}
```

The mapping template supplied here translates the `greeter` query string parameter to the `greeter` property of the JSON payload. This is necessary because input to a Lambda function in the Lambda function must be expressed in the body. You could use JSON string of the map (for example, `{"greeter": "'john'"}`) as the `request-template` input value to the `put-integration` command. However, using the file input avoids the difficult, and sometimes impossible, quote-escaping that is required to stringify a JSON object.

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.

The 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
```

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'
```

Compared to [the setup for the Lambda proxy integration \(p. 129\)](#), it is much more involved to set up a Lambda custom integration.

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 \(p. 139\)](#)
- [Handle Custom Lambda Errors in API Gateway \(p. 141\)](#)

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 Lambda function (Node.js 4.3 and later).

```
exports.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": [  
        "exports.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** configuration editor.

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 "Invalid*" --region us-west-2
```

Make sure that you also set up the corresponding error code (400) on the method request. Otherwise, API Gateway throws an invalid configuration error response at runtime.

At runtime, API Gateway matches the error message against the pattern of the regular expression on the `selectionPattern` property. When 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.

Note

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 API Gateway REST API, call the `integrationresponse:update` operation to replace the `/selectionPattern` path value with the specified regex expression of the `Malformed*` pattern.

```
PATCH /restapis/z0vprf0mdh/resources/x3o5ih/methods/GET/integration/responses/400 HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170513T044831Z  
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170513/us-west-2/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature=4b4e4881a727bc999799be5110facfcf7ec81d025afc1a4386cf440b4efc80de  
Cache-Control: no-cache  
Postman-Token: 04794263-29f2-f336-fa16-5653a7f3281d  
  
{  
  "patchOperations" : [ {  
    "op" : "replace",  
    "path" : "/selectionPattern",  
    "value" : "Malformed*"  
  } ]  
}
```

To set the `selectionPattern` expression using the API Gateway console, type 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 Lambda function (Node.js 4.3 or later) returns a custom error:

```
exports.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 body-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 Swagger 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.

With the proxy integration, the setup is simple. You only need to set the HTTP method and the HTTP endpoint URL, according to the backend requirements, if you are not concerned with content encoding or caching.

With the custom integration, the 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 \(p. 143\)](#)
- [Set up HTTP Custom Integrations in API Gateway \(p. 146\)](#)

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 a Swagger 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 [Build an API with HTTP Proxy Integration \(p. 39\)](#).

The following Swagger API definition file shows an example of an API with a proxy resource that is integrated with the [PetStore](#) website.

```
{
  "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 [Set up Data Mappings. \(p. 162\)](#)

Set up API Gateway Private Integrations

The API Gateway private integration makes it simple to expose your HTTP/HTTPS resources behind 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 for open access or controlled access. You can do this by using IAM permissions, a Lambda authorizer, or an Amazon Cognito user pool.

The private integration uses an API Gateway resource of `VpcLink` to encapsulate connections between API Gateway and targeted VPC resources. As an owner of a VPC resource, you are responsible for creating a network load balancer in your VPC and adding a VPC resource as a target of a network load balancer's listener. As an API developer, to set up an API with the private integration, you are responsible for creating a `VpcLink` targeting specified network load balancers and then treating the `VpcLink` as an effective integration endpoint.

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

- [Set up a Network Load Balancer for API Gateway Private Integrations \(p. 147\)](#)
- [Grant Permissions to Create a VPC Link \(p. 147\)](#)
- [Set up an API Gateway API with Private Integrations Using the API Gateway Console \(p. 148\)](#)
- [Set up an API Gateway API with Private Integrations Using the AWS CLI \(p. 148\)](#)
- [Set up API with Private Integrations Using Swagger \(p. 151\)](#)

Set up a Network Load Balancer for API Gateway Private Integrations

The following procedure outlines the steps to set up a network load balancer for API Gateway private integrations using the Amazon EC2 console and provides references for detailed instructions for each step.

To create a network load balancer for private integration using the API Gateway console

1. Sign in to the Amazon EC2 console at <https://console.aws.amazon.com/ec2/> and choose a region; for example, us-east-1, on the navigation bar.
2. Set up a web server on an Amazon EC2 instance. For an example setup, see [Installing a LAMP Web Server on Amazon Linux](#).
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](#).

After the network load balancer is created, note its ARN. 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.

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 and update a vpcLink

1. Create an IAM policy similar to the following:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "ec2:CreateVpcEndpointServiceConfiguration",  
                "ec2:DeleteVpcEndpointServiceConfigurations",  
                "ec2:DescribeVpcEndpointServiceConfigurations",  
                "ec2:ModifyVpcEndpointServicePermissions"  
            ],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "elasticloadbalancing:DescribeLoadBalancers"  
            ],  
            "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 [Build an API with API Gateway Private Integration \(p. 74\)](#).

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 \(p. 147\)](#) to install the VPC resource, create a NLB, set the VPC resource as a target of the network load balancer.

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 \(p. 147\)](#).

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.

For this discussion, we assume the ARN of the network load balancer is `arn:aws:elasticloadbalancing:us-east-1:123456789012:loadbalancer/net/my-vpclink-test-nlb/1f8df693cd094a72`.

```
aws apigateway create-vpc-link \
--name my-test-vpc-link \
--target-arns arn:aws:elasticloadbalancing:us-east-1:123456789012:loadbalancer/net/
my-vpclink-test-nlb/1f8df693cd094a72 \
--endpoint-url https://apigateway.us-east-1.amazonaws.com \
--region us-east-1
```

If the AWS configuration uses `us-east-1` as the default region, you can skip the `endpoint-url` and `region` parameters in the preceding input.

The preceding command immediately returns the following response, acknowledging the receipt of the request, and showing the `PENDING` status for the `VpcLink` being created.

```
{
  "status": "PENDING",
  "targetArns": [
    "arn:aws:elasticloadbalancing:us-east-1:123456789012:loadbalancer/net/my-
vpclink-test-nlb/1f8df693cd094a72"
  ],
  "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"
```

Only after the `VpcLink` is created successfully are we ready to create the 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'
```

We have dropped the input parameters of `endpoint-url` and `region` to use the default region as specified in the AWS configuration.

Note the `RestApi`'s `id` value in the returned result. In this example, we assume it is `6j4m3244we`. You need this value to perform further operations on the API, including setting up methods and integrations.

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 6j4m3244we
```

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 6j4m3244we \
--resource-id skpp60rab7 \
--http-method GET \
--authorization-type "NONE"
```

To use the IAM permissions, a Lambda authorizer, or an Amazon Cognito user pool to authenticate the caller, set the `authorization-type` to `AWS_IAM`, `CUSTOM`, or `COGNITO_USER_POOLS`, respectively.

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 6j4m3244we \
--resource-id skpp60rab7 \
--uri 'http://myApi.example.com' \
--http-method GET \
```

```
--type HTTP_PROXY \
--integration-http-method GET \
--connection-type VPC_LINK \
--connection-id gim7c3
```

For a private integration, you must 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.

If successful, the command returns the following output:

```
{
  "passthroughBehavior": "WHEN_NO_MATCH",
  "timeoutInMillis": 29000,
  "connectionId": "gim7c3",
  "uri": "http://myApi.example.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 6j4m3244we \
--resource-id skpp60rab7 \
--uri 'http://myApi.example.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 6j4m3244we \
--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.

Using a stage variable to set the `connectionId` value has the advantage of having the same API integrated with different `VpcLinks` by resetting the stage variable value. This is useful for switching your API to a different VPC link to migrate to a different network load balancer or a different VPC.

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 \(p. 45\)](#).

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 6j4m3244we \
--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 6j4m3244we \
--stage-name test \
--patch-operations op=replace,path='/variables/vpcLinkId',value='asf9d7'
```

To test the API, invoke it using the following cURL command:

```
curl -X GET https://6j4m3244we.execute-api.us-east-1.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 Swagger

You can set up an API with the private integration by importing the API Swagger file. The settings are similar to the Swagger 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 API Swagger file shows an example of an API with a VPC link as referenced by a stage variable (`#{stageVariables.vpcLinkId}`):

```
{
  "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": {
            "$ref": "#/definitions/Empty"
          }
        }
      }
    },
    "x-amazon-apigateway-integration": {
      "responses": {
        "default": {
          "statusCode": "200"
        }
      },
      "uri": "http://myApi.example.com",
      "passthroughBehavior": "when_no_match",
      "connectionType": "VPC_LINK",
      "connectionId": "${stageVariables.vpcLinkId}",
      "httpMethod": "GET",
      "type": "http_proxy"
    }
  }
},
"definitions": {
  "Empty": {
    "type": "object",
    "title": "Empty Schema"
  }
}
}
]

```

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 [Build an API Gateway API from an Example \(p. 9\)](#).

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.

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 \(p. 153\)](#)

Enable Mock Integration Using the API Gateway Console

You must have the method available in API Gateway. Follow the instructions in [Build an API with HTTP Custom Integration \(p. 44\)](#).

1. Choose an API resource and create a method. In the method **Setup** pane, choose **Mock** for **Integration type**, and then choose **Save**.
2. Choose **Method Request** from **Method Execution**. Expand **URL Query String Parameters**. Choose **Add query string** to add a scope query parameter. This determines if the caller is internal or otherwise.
3. Choose **Integration Request** from **Method Execution**. Expand **Body Mapping Templates**. Choose or add an **application/json** mapping template. Type the following in the template editor:

```
{
    #if( $input.params('scope') == "internal" )
        "statusCode": 200
    #else
```

```

        "statusCode": 500
    #end
}

```

Choose **Save**.

4. Choose **Integration Response** from **Method Execution**. Expand the **200** response and then the **Body Mapping Templates** section. Choose or add an application/json mapping template and type the following response body mapping template in the template editor.

```

{
    "statusCode": 200,
    "message": "Go ahead without me"
}

```

Choose **Save**.

5. Scroll to **Integration Response**. Choose **Add integration response** to add a 500 response. Type `5\d{2}` in **HTTP status regex**. Expand **Body Mapping Templates** and choose **Add mapping template**. Type `application/json` for **Content-Type** and then choose the check mark icon to save the setting. In the template editor, type the following integration response body mapping template:

```

{
    "statusCode": 500,
    "message": "The invoked method is not supported on the API resource."
}

```

Choose **Save**.

6. Choose **Test** from **Method Execution**. Do the following:

- a. Type `internal` under **scope**. 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. Type `public` under **scope** 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.

Set 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 `$context` variables, see [Accessing the \\$context Variable \(p. 191\)](#). For more information about `$stageVariables`, see [Accessing the \\$stageVariables Variable \(p. 198\)](#). For more information about method request parameters, see [Request parameters accessible by a mapping template \(p. 187\)](#).

Topics

- [Gateway Responses in API Gateway \(p. 156\)](#)
- [Gateway Response Types \(p. 156\)](#)
- [Set up a Gateway Response Using the API Gateway Console \(p. 159\)](#)
- [Set up a Gateway Response Using the API Gateway REST API \(p. 161\)](#)
- [Set up Gateway Response Customization in Swagger \(p. 161\)](#)

Gateway Responses in API Gateway

A gateway response is identified by a response type 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 Swagger, a `GatewayResponse` instance is described by the [x-amazon-apigateway-gateway-responses.gatewayResponse \(p. 493\)](#) extension.

To enable a gateway response, you set up a gateway response for a [supported response type \(p. 156\)](#) at the API level. Whenever API Gateway returns a response of the 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 using the API Gateway console and the API Gateway REST API.

Gateway Response Types

API Gateway exposes the following gateway responses for customization by API developers.

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.
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.
ACCESS_DENIED	403	The gateway response for authorization failure; for example, when access is denied by a custom or Amazon Cognito authorizer. If the response type is unspecified, this response defaults to the <code>DEFAULT_4XX</code> type.

Gateway response type	Default status code	Description
API_CONFIGURATION_ERROR	500	The gateway response for invalid API configuration, including invalid endpoint address submitted, Base64 decoding failed on binary data when binary support is enacted, or integration response mapping cannot 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		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.
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.
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.

Gateway response type	Default status code	Description
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.
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 the DEFAULT_4XX type.

Gateway response type	Default status code	Description
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 <code>DEFAULT_4XX</code> type.
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 <code>DEFAULT_4XX</code> 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 <code>DEFAULT_4XX</code> type.

Set up a Gateway Response Using the API Gateway Console

To customize a gateway response using the API Gateway console

1. Sign in to the API Gateway console.
2. Choose your existing API or create a new one.
3. Expand the API in the primary navigation pane and choose **Gateway Responses** under the API.
4. In the **Gateway Responses** pane, choose a response type. In this walkthrough, we use **Missing Authentication Token (403)** 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 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 **Body Mapping Templates**, leave `application/json` for **Content Type** and type the following body mapping template in the **Body Mapping Template** 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**.
9. Deploy the API to a new or existing stage.
10. Test it 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` is not 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: a2be05a4-2ee5-11e7-bbf2-df131ec50ae6
Access-Control-Allow-Origin: a.b.c
x-amzn-ErrorResponse: 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-Mk93KgoxnfzDlTUh3fImzA==

{
  "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, specifying a desired `responseType` in the URL path parameter and supplying in the request payload the `statusCode`, `responseParameters` and `responseTemplates` mappings.
2. To update part of a `GatewayResponse` instance, call the `gatewayresponse:update` action, specifying a desired `responseType` in the URL path parameter and supplying in the request payload desired individual `GatewayResponse` properties, for example, the `responseParameters` or the `responseTemplates` mapping.

Set up Gateway Response Customization in Swagger

You can use the `x-amazon-apigateway-gateway-responses` extension at the API root level to customize gateway responses in Swagger. The following Swagger 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            },\n            \"$context.error.responseType\": \"$stage\", \"$stageVariables.a\": \"$stageVariables.a\"\n        }\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.

Set up API Gateway Request and Response Data Mappings

Topics

- [Set up Request and Response Data Mappings Using the API Gateway Console \(p. 162\)](#)
- [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#)
- [Amazon API Gateway API Request and Response Data Mapping Reference \(p. 187\)](#)
- [API Gateway Mapping Template Reference \(p. 191\)](#)

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 \(p. 123\)](#).

1. With the method selected in the **Resources** pane, in the **Method Execution** pane, choose **Integration Request**.
2. 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 the arrow next to **URL Path Parameters**, **URL Query String Parameters**, or **HTTP 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**, type 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.header.parameter-name** for a header parameter named *parameter-name* as defined in the **Method Request** page.
3. Alternatively, you can set a literal string value (enclosed by a pair of single quotes) to an integration header.
 - d. Choose **Create**. (To delete a path parameter, query string parameter, or header parameter, choose **Cancel** or **Remove** next to the parameter you want to delete.)
3. In the **Body Mapping Templates** area, 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 \(p. 190\)](#).

4. To define a mapping template for an incoming request, choose **Add mapping template** under **Content-Type**. Type a content type (e.g., `application/json`) in the input text box and then choose the check mark icon to save the input. Then, type the mapping template manually or choose **Generate template** to create one from a model template. For more information, see [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).
5. 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 back end, 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 Response** from the **Method Execution** page.
 - a. In the **Method Execution** pane, choose **Integration Response**. Choose either the arrow next to **200** to specify settings for a 200 HTTP response code from the method, or choose **Add integration 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), type 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, type "`.*Invalid request.*`" as the **Lambda error regex** expression. On the other hand, to return 400 Bad Request for all unmapped error messages from Lambda, type "`(\\n|.)+`" in **Lambda error regex**. This last regular expression can be used for the default error response of an API.

Note

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**, choose the HTTP response status code you defined in the **Method Response** page.
- d. For **Header Mappings**, for each header you defined for the HTTP response status code in the **Method Response** page, specify a mapping value by choosing **Edit**. For **Mapping value**, use the format `integration.response.header.header-name` where `header-name` is the name of a 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 an **Timestamp** entry and the associated **Mapping value** should be set to `integration.response.header.Date`.
- e. In the **Template Mappings** area, next to **Content type**, choose **Add**. In the **Content type** box, type the content type of the data that will be passed from the Lambda function, HTTP proxy, or AWS service proxy to the method. Choose **Update**.
- f. Select **Output passthrough** if you want the method to receive, but not modify, the data from the Lambda function, HTTP proxy, or AWS service proxy.
- g. If **Output passthrough** is cleared, for **Output mapping**, specify the output mapping template you want the Lambda function, HTTP proxy, or AWS service proxy to use to send data to the method. You can either type the mapping template manually or choose a model from **Generate template from model**.
- h. Choose **Save**.

Create Models and Mapping Templates for Request and Response Mappings

In API Gateway, an 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.

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](#). You must define the model in order to have API Gateway to generate a SDK or to enable basic request validation for your API. You do not have to define any model to create a mapping template. However, a model can help you create a template because API Gateway will generate a template blueprint based on a provided model.

The section explains how to map the API request and response payload using models and mapping templates.

Topics

- [Models \(p. 165\)](#)
- [Mapping Templates \(p. 168\)](#)
- [Tasks for Models and Mapping Templates \(p. 170\)](#)
- [Create a Model in API Gateway \(p. 170\)](#)

- [View a List of Models in API Gateway \(p. 171\)](#)
- [Delete a Model in API Gateway \(p. 171\)](#)
- [Photos Example \(API Gateway Models and Mapping Templates\) \(p. 171\)](#)
- [News Article Example \(API Gateway Models and Mapping Templates\) \(p. 175\)](#)
- [Sales Invoice Example \(API Gateway Models and Mapping Templates\) \(p. 178\)](#)
- [Employee Record Example \(API Gateway Models and Mapping Templates\) \(p. 182\)](#)

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 describes a sample data describing the fruit or vegetable inventory in the produce department of a likely supermarket:

Suppose we have an API for managing fruit and vegetable inventory in the produce department of a supermarket. When a manager queries the backend for the current inventory, the server sends back the following response payload:

```
{  
    "department": "produce",  
    "categories": [  
        "fruit",  
        "vegetables"  
    ],  
    "bins": [  
        {  
            "category": "fruit",  
            "type": "apples",  
            "price": 1.99,  
            "unit": "pound",  
            "quantity": 232  
        },  
        {  
            "category": "fruit",  
            "type": "bananas",  
            "price": 0.19,  
            "unit": "each",  
            "quantity": 112  
        },  
        {  
            "category": "vegetables",  
            "type": "carrots",  
            "price": 1.29,  
            "unit": "bag",  
            "quantity": 57  
        }  
    ]  
}
```

The JSON object has three properties

- The `department` property has a string value (`produce`).
- The `categories` property is an array of two strings: `fruit` and `vegetables`.
- The `bins` property is an array of objects, each having the string- or number-valued properties of `category`, `type`, `price`, `unit` and `quantity`.

We can use the following JSON Schema to define the model for the above data:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "GroceryStoreInputModel",
  "type": "object",
  "properties": {
    "department": { "type": "string" },
    "categories": {
      "type": "array",
      "items": { "type": "string" }
    },
    "bins": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "category": { "type": "string" },
          "type": { "type": "string" },
          "price": { "type": "number" },
          "unit": { "type": "string" },
          "quantity": { "type": "integer" }
        }
      }
    }
  }
}
```

In the preceding example model:

- The `$schema` object represents a valid JSON Schema version identifier. In this example, it refers to JSON Schema, draft v4.
- The `title` object is a human-readable identifier for the model. In this example, it is `GroceryStoreInputModel`.
- The top-level, or root, construct in the JSON data is an object.
- The root object in the JSON data contains `department`, `categories`, and `bins` properties.
- The `department` property is a string object in the JSON data.
- The `categories` property is an array in the JSON data. The array contains string values in the JSON data.
- The `bins` property is an array in the JSON data. The array contains objects in the JSON data. Each of these objects in the JSON data contains a `category` string, a `type` string, a `price` number, a `unit` string, and a `quantity` integer (a number without a fraction or exponent part).

Alternatively, you could include part of this schema, for example, the item definition of the `bins` array, in a separate section of the same file and use the `$ref` primitive to reference this reusable definition in other parts of the schema. Using `$ref`, the above model definition file can be expressed as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "GroceryStoreInputModel",
  "type": "object",
  "properties": {
    "department": { "type": "string" },
    "categories": {
      "type": "array",
      "items": { "type": "string" }
    },
    "bins": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/Bin"
      }
    }
  }
}
```

```

        }
    },
    "definitions": {
        "Bin" : {
            "type": "object",
            "properties": {
                "category": { "type": "string" },
                "type": { "type": "string" },
                "price": { "type": "number" },
                "unit": { "type": "string" },
                "quantity": { "type": "integer" }
            }
        }
    }
}

```

The `definitions` section contains the schema definition of the `Bin` item that is referenced in the `bins` array with `"ref": "#/definitions/Bin"`. Using reusable definitions this way makes your model definition easier to read.

In addition, you can also reference another model schema defined in an external model file by setting that model's URL as the value of the `$ref` property: `"$ref": "https://apigateway.amazonaws.com/restapis/{restapi_id}/models/{model_name}"`. For example, supposed you have the following full-fledged model named `Bin2` created under an API with an identifier of `fugvwdxtri`:

```

{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "GroceryStoreInputModel",
    "type": "object",
    "properties": {
        "Bin" : {
            "type": "object",
            "properties": {
                "category": { "type": "string" },
                "type": { "type": "string" },
                "price": { "type": "number" },
                "unit": { "type": "string" },
                "quantity": { "type": "integer" }
            }
        }
    }
}

```

You can then reference it from the `GroceryStoreInputModel` from the same API, as shown as follows:

```

{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "GroceryStoreInputModel",
    "type": "object",
    "properties": {
        "department": { "type": "string" },
        "categories": {
            "type": "array",
            "items": { "type": "string" }
        },
        "bins": {
            "type": "array",
            "items": {
                "$ref": "https://apigateway.amazonaws.com/restapis/fugvwdxtri/models/Bin2"
            }
        }
    }
}

```

```
    }
}
```

The referencing and referenced models must be from the same API.

The examples do not use advanced JSON Schema features, such as specifying required items; minimum and maximum allowed string lengths, numeric values, and array item lengths; regular expressions; and more. For more information, see [Introducing JSON](#) and [JSON schema draft 4](#).

For more complex JSON data formats and their models, see the following examples:

- [Input Model \(Photos Example\) \(p. 172\)](#) and [Output Model \(Photos Example\) \(p. 174\)](#) in the [Photos Example \(p. 171\)](#)
- [Input Model \(News Article Example\) \(p. 175\)](#) and [Output Model \(News Article Example\) \(p. 177\)](#) in the [News Article Example \(p. 175\)](#)
- [Input Model \(Sales Invoice Example\) \(p. 179\)](#) and [Output Model \(Sales Invoice Example\) \(p. 181\)](#) in the [Sales Invoice Example \(p. 178\)](#)
- [Input Model \(Employee Record Example\) \(p. 183\)](#) and [Output Model \(Employee Record Example\) \(p. 185\)](#) in the [Employee Record Example \(p. 182\)](#)

To experiment with models in API Gateway, follow the instructions in [Map Response Payload \(p. 62\)](#), specifically [Step 1: Create Models \(p. 64\)](#).

Mapping Templates

When the backend returns the query results (shown in the [Models \(p. 165\)](#) section), the manager of the produce department may be interested in reading them as follows:

```
{
  "choices": [
    {
      "kind": "apples",
      "suggestedPrice": "1.99 per pound",
      "available": 232
    },
    {
      "kind": "bananas",
      "suggestedPrice": "0.19 per each",
      "available": 112
    },
    {
      "kind": "carrots",
      "suggestedPrice": "1.29 per bag",
      "available": 57
    }
  ]
}
```

To enable this, we need to provide API Gateway a mapping template to translate the data from the backend format. The following mapping template will do just that.

```
#set($inputRoot = $input.path('$'))
{
  "choices": [
#foreach($elem in $inputRoot.bins)
  {
```

```

        "kind": "$elem.type",
        "suggestedPrice": "$elem.price per $elem.unit",
        "available": $elem.quantity
    }#if($foreach.hasNext),#end

#endif
]
}

```

Let us now examine some details of the preceding output mapping template:

- The `$inputRoot` variable represents the root object in the original JSON data from the previous section. The variables in an output mapping template map to the original JSON data, not the desired transformed JSON data schema.
- The `choices` array in the output mapping template is mapped from the `bins` array with the root object in the original JSON data (`$inputRoot.bins`).
- In the output mapping template, each of the objects in the `choices` array (represented by `$elem`) are mapped from the corresponding objects in the `bins` array within the root object in the original JSON data.
- In the output mapping template, for each of objects in the `choices` object, the values of the `kind` and `available` objects (represented by `$elem.type` and `$elem.quantity`) are mapped from the corresponding values of the `type` and `value` objects in each of the objects in the original JSON data's `bins` array, respectively.
- In the output mapping template, for each of objects in the `choices` object, the value of the `suggestedPrice` object is a concatenation of the corresponding value of the `price` and `unit` objects in each of the objects in the original JSON data, respectively, with each value separated by the word `per`.

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. As an API developer, you know the data formats at both the front and backends. That knowledge can guide you to define the necessary mappings without ambiguity.

To have an SDK generated for the API, the above data will be returned as a language-specific object. For strongly typed languages, such as Java, Objective-C or Swift, the object corresponds to a user-defined data type (UDT). API Gateway will create such a UDT if you provide it with a data model. For the method response example above, you can define the following payload model in the integration response:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "GroceryStoreOutputModel",
    "type": "object",
    "properties": {
        "choices": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "kind": { "type": "string" },
                    "suggestedPrice": { "type": "string" },
                    "available": { "type": "integer" }
                }
            }
        }
    }
}
```

In this model, the JSON schema is expressed as follows:

- The `$schema` object represents a valid JSON Schema version identifier. In this example, it refers to [JSON Schema, draft v4](#).
- The `title` object is a human-readable identifier for the model. In this example, it is `GroceryStoreOutputModel`.
- The top-level, or root, construct in the JSON data is an object.
- The root object in the JSON data contains an array of objects.
- Each object in the array of objects contains a `kind` string, a `suggestedPrice` string, and an `available` integer (a number without a fraction or exponent part).

With this model, you can call an SDK to retrieve the `kind`, `suggestedPrice` and `available` property values by reading the `GroceryStoreOutputModel.kind`, `GroceryStoreOutputModel.suggestedPrice` and `GroceryStoreOutputModel.available` properties, respectively. If no model is provided, API Gateway will use the [Empty model](#) to create a default UDT. In this case, you will not be able to read these properties using a strongly-typed SDK.

To explore more complex mapping templates, see the following examples:

- [Input Mapping Template \(Photos Example\) \(p. 173\)](#) and [Output Mapping Template \(Photos Example\) \(p. 174\)](#) in the [Photos Example \(p. 171\)](#)
- [Input Mapping Template \(News Article Example\) \(p. 176\)](#) and [Output Mapping Template \(News Article Example\) \(p. 177\)](#) in the [News Article Example \(p. 175\)](#)
- [Input Mapping Template \(Sales Invoice Example\) \(p. 180\)](#) and [Output Mapping Template \(Sales Invoice Example\) \(p. 182\)](#) in the [Sales Invoice Example \(p. 178\)](#)
- [Input Mapping Template \(Employee Record Example\) \(p. 184\)](#) and [Output Mapping Template \(Employee Record Example\) \(p. 186\)](#) in the [Employee Record Example \(p. 182\)](#)

To experiment with mapping templates in API Gateway, follow the instructions in [Map Response Payload \(p. 62\)](#), specifically [Step 5: Set up and Test the Methods \(p. 68\)](#).

Tasks for Models and Mapping Templates

For additional things you can do with models and mapping templates, see the following:

- [Create a Model \(p. 170\)](#)
- [View a List of Models \(p. 171\)](#)
- [Delete a Model \(p. 171\)](#)

Create a Model in API Gateway

Use the API Gateway console to create a model for an API.

Topics

- [Prerequisites \(p. 170\)](#)
- [Create a Model With the API Gateway Console \(p. 171\)](#)

Prerequisites

- You must have an API available in API Gateway. Follow the instructions in [Creating an API in Amazon API Gateway \(p. 81\)](#).

Create a Model With the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API where you want to create the model, choose **Models**.
3. Choose **Create**.
4. For **Model Name**, type a name for the model.
5. For **Content Type**, type the model's content type (for example, `application/json` for JSON).
6. (Optional) For **Model description**, type a description for the model.
7. For **Model schema**, type the model's schema. For more information about model schemas, see [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).
8. Choose **Create model**.

View a List of Models in API Gateway

Use the API Gateway console to view a list of models.

Topics

- [Prerequisites \(p. 171\)](#)
- [View a List of Models with the API Gateway Console \(p. 171\)](#)

Prerequisites

- You must have at least one model in API Gateway. Follow the instructions in [Create a Model \(p. 170\)](#).

View a List of Models with the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API, choose **Models**.

Delete a Model in API Gateway

Use the API Gateway console to delete a model.

Warning

Deleting a model may cause part or all of the corresponding API to become unusable by API callers. Deleting a model cannot be undone.

Delete a Model with the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API for the model, choose **Models**.
3. In the **Models** pane, choose the model you want to delete, and then choose **Delete Model**.
4. When prompted, choose **Delete**.

Photos Example (API Gateway Models and Mapping Templates)

The following sections provide examples of models and mapping templates that could be used for a sample photo API in API Gateway. For more information about models and mapping templates in API Gateway, see [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

Topics

- [Original Data \(Photos Example\) \(p. 172\)](#)
- [Input Model \(Photos Example\) \(p. 172\)](#)
- [Input Mapping Template \(Photos Example\) \(p. 173\)](#)
- [Transformed Data \(Photos Example\) \(p. 173\)](#)
- [Output Model \(Photos Example\) \(p. 174\)](#)
- [Output Mapping Template \(Photos Example\) \(p. 174\)](#)

Original Data (Photos Example)

The following is the original JSON data for the photos example:

```
{
  "photos": {
    "page": 1,
    "pages": "1234",
    "perpage": 100,
    "total": "123398",
    "photo": [
      {
        "id": "12345678901",
        "owner": "234567890@A12",
        "secret": "abc123d456",
        "server": "1234",
        "farm": 1,
        "title": "Sample photo 1",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0
      },
      {
        "id": "23456789012",
        "owner": "34567890@B23",
        "secret": "bcd234e567",
        "server": "2345",
        "farm": 2,
        "title": "Sample photo 2",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0
      }
    ]
  }
}
```

Input Model (Photos Example)

The following is the input model that corresponds to the original JSON data for the photos example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosInputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "object",
      "properties": {
        "page": { "type": "integer" },
        "pages": { "type": "string" },
        "perpage": { "type": "integer" },
        "total": { "type": "string" }
      }
    }
  }
}
```

```
    "perpage": { "type": "integer" },
    "total": { "type": "string" },
    "photo": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "id": { "type": "string" },
                "owner": { "type": "string" },
                "secret": { "type": "string" },
                "server": { "type": "string" },
                "farm": { "type": "integer" },
                "title": { "type": "string" },
                "ispublic": { "type": "integer" },
                "isfriend": { "type": "integer" },
                "isfamily": { "type": "integer" }
            }
        }
    }
}
```

Input Mapping Template (Photos Example)

The following is the input mapping template that corresponds to the original JSON data for the photos example:

```
#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": "$elem.owner",
        "secret": "$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
    ]
}
}
```

Transformed Data (Photos Example)

The following is one example of how the original photos example JSON data could be transformed for output:

```
{
    "photos": [
```

```
{
  "id": "12345678901",
  "owner": "23456789@A12",
  "title": "Sample photo 1",
  "ispublic": 1,
  "isfriend": 0,
  "isfamily": 0
},
{
  "id": "23456789012",
  "owner": "34567890@B23",
  "title": "Sample photo 2",
  "ispublic": 1,
  "isfriend": 0,
  "isfamily": 0
}
]
```

Output Model (Photos Example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosOutputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string" },
          "owner": { "type": "string" },
          "title": { "type": "string" },
          "ispublic": { "type": "integer" },
          "isfriend": { "type": "integer" },
          "isfamily": { "type": "integer" }
        }
      }
    }
  }
}
```

Output Mapping Template (Photos 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 = $input.path('$'))
{
  "photos": [
    #foreach($elem in $inputRoot.photos.photo)
    {
      "id": "$elem.id",
      "owner": "$elem.owner",
      "title": "$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 [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

Topics

- [Original Data \(News Article Example\) \(p. 175\)](#)
- [Input Model \(News Article Example\) \(p. 175\)](#)
- [Input Mapping Template \(News Article Example\) \(p. 176\)](#)
- [Transformed Data \(News Article Example\) \(p. 177\)](#)
- [Output Model \(News Article Example\) \(p. 177\)](#)
- [Output Mapping Template \(News Article Example\) \(p. 177\)](#)

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

#endif
],
"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 [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

Topics

- [Original Data \(Sales Invoice Example\) \(p. 178\)](#)
- [Input Model \(Sales Invoice Example\) \(p. 179\)](#)
- [Input Mapping Template \(Sales Invoice Example\) \(p. 180\)](#)
- [Transformed Data \(Sales Invoice Example\) \(p. 181\)](#)
- [Output Model \(Sales Invoice Example\) \(p. 181\)](#)
- [Output Mapping Template \(Sales Invoice Example\) \(p. 182\)](#)

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 [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#).

Topics

- [Original Data \(Employee Record Example\) \(p. 182\)](#)
- [Input Model \(Employee Record Example\) \(p. 183\)](#)
- [Input Mapping Template \(Employee Record Example\) \(p. 184\)](#)
- [Transformed Data \(Employee Record Example\) \(p. 185\)](#)
- [Output Model \(Employee Record Example\) \(p. 185\)](#)
- [Output Mapping Template \(Employee Record Example\) \(p. 186\)](#)

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",  
            "Email": "jane.doe@example.com",  
            "Phone": "555-1234",  
            "Address": "123 Main St",  
            "City": "Anytown",  
            "State": "CA",  
            "Zip": "90210",  
            "Country": "US",  
            "HireDate": "2023-01-01",  
            "LastUpdate": "2023-01-01",  
            "IsEnabled": true  
        }  
    }  
}
```

```

    "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:

```

    "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 \(p. 191\)](#), [stage \(p. 198\)](#), or [util \(p. 198\)](#) 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 \(p. 393\)](#).

Topics

- [Map Method Request Data to Integration Request Parameters \(p. 187\)](#)
- [Map Integration Response Data to Method Response Headers \(p. 188\)](#)
- [Map Request and Response Payloads between Method and Integration \(p. 189\)](#)
- [Integration Passthrough Behaviors \(p. 190\)](#)

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.

Integration request data mapping expressions

Mapped data source	Mapping expression
Method request path	method.request.path. <i>PARAM_NAME</i>
Method request query string	method.request.querystring. <i>PARAM_NAME</i>
Method request header	method.request.header. <i>PARAM_NAME</i>
Method request body	method.request.body
Method request body (JsonPath)	method.request.body. <i>JSONPath_EXPRESSION</i>
Stage variables	stageVariables. <i>VARIABLE_NAME</i>
Context variables	context. <i>VARIABLE_NAME</i> that must be one of the supported context variables (p. 191) .
Static value	' <i>STATIC_VALUE</i> '. The <i>STATIC_VALUE</i> is a string literal and must be enclosed within a pair of single quotes.

Here, `PARAM_NAME` is the name of a method request parameter of the given parameter type. 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. However, the `$. prefix is omitted in this syntax.`

Example mappings from method request parameter in Swagger

The following example shows a Swagger snippet that maps 1) the method request's header, named `methodRequestPathParam`, into the integration request path parameter, named `integrationPathParam`; 2) the 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.querystring.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 Swagger

The following example shows a Swagger 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	<code>integration.response.header.PARAM_NAME</code>
Integration response body	<code>integration.response.body</code>
Integration response body (JsonPath)	<code>integration.response.body.JSONPath_EXPRESSION</code>

Mapped Data Source	Mapping expression
Stage variable	<code>stageVariables.VARIABLE_NAME</code>
Context variable	<code>context.VARIABLE_NAME</code> that must be one of the supported context variables (p. 191).
Static value	<code>'STATIC_VALUE'</code> . The <code>STATIC_VALUE</code> is a string literal and must be enclosed within a pair of single quotes.

Example data mapping from integration response in Swagger

The following example shows a Swagger 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",
}
...
```

Map Request and Response Payloads between Method and Integration

API Gateway uses [Velocity Template Language \(VTL\)](#) engine to process body [mapping templates](#) (p. 168) 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](#) (p. 122), API Gateway passes entire request through to backend and you do not have any 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](#) (p. 162), and the Content Type header that a client set in the incoming request. 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_TEMPLATE	NEVER
<code>None</code> (default to <code>application/json</code>)	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
<code>application/json</code>	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
<code>application/xml</code>	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.

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_TEMPLATE	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 Reference

Amazon API Gateway defines a set of variables and functions for working with models and mapping templates. This document describes those functions and provides examples for working with input payloads.

Topics

- [Accessing the \\$context Variable \(p. 191\)](#)
- [Accessing the \\$input Variable \(p. 195\)](#)
- [Accessing the \\$stageVariables Variable \(p. 198\)](#)
- [Accessing the \\$util Variable \(p. 198\)](#)

Accessing the \$context Variable

The \$context variable holds all the contextual information of your API call.

\$context Variable Reference

Parameter	Description
\$context.apiId	The identifier API Gateway assigns to your API.
\$context.authorizer.claims. <i>property</i>	A property of the claims returned from the Amazon Cognito user pool after the method caller is successfully authenticated. 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) Lambda function.

Parameter	Description
<code>\$context.authorizer.property</code>	The stringified value of the specified key-value pair of the context map returned from an API Gateway Lambda authorizer Lambda function. For example, if the authorizer returns the following context map:
	<pre>"context" : { "key": "value", "numKey": 1, "boolKey": true }</pre>
	calling <code>\$context.authorizer.key</code> returns the "value" string, calling <code>\$context.authorizer.numKey</code> returns the "1" string, and calling <code>\$context.authorizer.boolKey</code> returns the "true" string.
<code>\$context.httpMethod</code>	The HTTP method used. Valid values include: DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT.
<code>\$context.error.message</code>	A string of 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.
<code>\$context.error.messageString</code>	The quoted value of <code>\$context.error.message</code> , namely " <code>\$context.error.message</code> ".
<code>\$context.error.responseType</code>	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.
<code>\$context.extendedRequestId</code>	An automatically generated ID for the API call, which contains more useful information for debugging/troubleshooting.
<code>\$context.identity.accountId</code>	The AWS account ID associated with the request.
<code>\$context.identity.apiKey</code>	The API owner key associated with key-enabled API request.
<code>\$context.identity.apiKeyId</code>	The API key ID associated with the key-enabled API request
<code>\$context.identity.caller</code>	The principal identifier of the caller making the request.

Parameter	Description
<code>\$context.identity.cognitoAuthenticationProvider</code>	The Amazon Cognito authentication provider used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For information related to this and the other Amazon Cognito <code>\$context</code> variables, see Using Federated Identities in the <i>Amazon Cognito Developer Guide</i> .
<code>\$context.identity.cognitoAuthenticationType</code>	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
<code>\$context.identity.cognitoIdentityId</code>	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
<code>\$context.identity.cognitoIdentityPoolId</code>	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
<code>\$context.identity.sourceIp</code>	The source IP address of the TCP connection making the request to API Gateway.
<code>\$context.identity.user</code>	The principal identifier of the user making the request.
<code>\$context.identity.userAgent</code>	The User Agent of the API caller.
<code>\$context.identity.userArn</code>	The Amazon Resource Name (ARN) of the effective user identified after authentication.
<code>\$context.integrationLatency</code>	The integration latency in ms, available for access logging only.
<code>\$context.path</code>	The request path. For example, for the non-proxy request URI of <code>https://{{rest-api-id}.execute-api.{region}.amazonaws.com/{stage}/root/child}</code> , The <code>\$context.path</code> value is <code>/{stage}/root/child</code> .
<code>\$context.protocol</code>	The request protocol, for example, <code>HTTP/1.1</code> .
<code>\$context.requestId</code>	An automatically generated ID for the API call.
<code>\$context.requestTime</code>	The <code>CLF</code> -formatted request time (<code>dd/MMM/yyyy:HH:mm:ss +-hhmm</code>).
<code>\$context.requestTimeEpoch</code>	The <code>Epoch</code> -formatted request time.
<code>\$context.resourceId</code>	The identifier API Gateway assigns to your resource.

Parameter	Description
<code>\$context.resourcePath</code>	The path to your resource. For example, for the non-proxy request URI of <code>https://rest-api-id.execute-api.{region}.amazonaws.com/{stage}/root/child</code> , The <code>\$context.resourcePath</code> value is <code>/root/child</code> . For more information, see Build an API with HTTP Custom Integration (p. 44) .
<code>\$context.responseLength</code>	The response payload length, available for access logging only.
<code>\$context.responseLatency</code>	The response latency in ms, available for access logging only.
<code>\$context.status</code>	The response status, available for access logging only.
<code>\$context.stage</code>	The deployment stage of the API call (for example, Beta or Prod).

Example

You may want to use the `$context` variable if you're using AWS Lambda as the target backend that the API method calls. For example, you may want to perform two different actions depending on whether the stage is in Beta or in Prod.

Context Variables Template Example

The following example shows a mapping template to map context variables to an integration request payload:

```
{
  "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"
}
```

In the above example, the method is assumed to have an API key enabled. If API key is not required on the method request, `api_key` will be null.

For requests of the `AWS_IAM` authorization type, you can pass the authorized user information to the integration endpoint with `$context.identity.*` properties. For requests of the `COGNITO_USER_POOLS` authorization type, the authorized user information will also include `$context.identity.cognito*` and `$context.authorizer.claims.*` properties. For requests using a Lambda authorizer, you can pass `$context.authorizer.principalId` and other applicable `$context.authorizer.*` properties as additional authorized user context to the integration endpoint.

With a proxy integration, API Gateway passes the authorized identity information to the backend in the `requestContext.identity` object. You do not set up any mapping template and, instead, parse the input to the integration backend explicitly. The following shows an example of `requestContext` passed to a Lambda proxy integration endpoint when the authorization type is set to `AWS_IAM`.

```
{
  ...
  "requestContext": {
    "requestTime": "20/Feb/2018:22:48:57 +0000",
    "path": "/test/",
    "accountId": "123456789012",
    "protocol": "HTTP/1.1",
    "resourceId": "yx5mhem7ye",
    "stage": "test",
    "requestTimeEpoch": 1519166937665,
    "requestId": "3c3ecbaa-1690-11e8-ae31-8f39f1d24afdf",
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": "123456789012",
      "cognitoIdentityId": null,
      "caller": "AIDAJ.....4HCKVJZG",
      "sourceIp": "51.240.196.104",
      "accessKey": "IAM_user_access_key",
      "cognitoAuthenticationType": null,
      "cognitoAuthenticationProvider": null,
      "userArn": "arn:aws:iam::123456789012:user/alice",
      "userAgent": "PostmanRuntime/7.1.1",
      "user": "AIDAJ.....4HCKVJZG"
    },
    "resourcePath": "/",
    "httpMethod": "GET",
    "apiId": "qr2gd9cfmf"
  },
  ...
}
```

Accessing the `$input` Variable

The `$input` variable represents the input payload and parameters to be processed by your template. It provides four functions:

Function Reference

Variable and Function	Description
<code>\$input.body</code>	Returns the raw payload as a string.
<code>\$input.json(x)</code>	<p>This function evaluates a JSONPath expression and returns the results as a JSON string.</p> <p>For example, <code>\$input.json('\$.pets')</code> will return a JSON string representing the pets structure.</p> <p>For more information about JSONPath, see JSONPath or JSONPath for Java.</p>
<code>\$input.params()</code>	Returns a map of all the request parameters of your API call.

Variable and Function	Description
<code>\$input.params(x)</code>	Returns the value of a method request parameter from the path, query string, or header value (in that order) given a parameter name string <code>x</code> .
<code>\$input.path(x)</code>	Takes a JSONPath expression string (<code>x</code>) and returns an 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, <code>\$input.path('\$.pets').size()</code> For more information about JSONPath, see JSONPath or JSONPath for Java .

Examples

You may want to use the `$input` variable to get query strings and the request body with or without using models. You may also want to get the parameter and the payload, or a subsection of the payload, into your AWS Lambda function. The examples below show how to do this.

Example JSON Mapping Template

The following example shows how to use a mapping to read a name from the query string and then include the entire POST body in an element:

```
{
  "name" : "$input.params('name')",
  "body" : $input.json('$')
}
```

If the JSON input contains unescaped characters that cannot be parsed by JavaScript, a 400 response may be returned. Applying `$util.escapeJavaScript($input.json('$'))` above will ensure that the JSON input can be parsed properly.

Example Inputs Mapping Template

The following example shows how to pass a JSONPath expression to the `json()` method. You could also read a specific property of your request body object by using a period (`.`), followed by your property name:

```
{
  "name" : "$input.params('name')",
  "body" : $input.json('$.mykey')
}
```

If a method request payload contains unescaped characters that cannot be parsed by JavaScript, you may get 400 response. In this case, you need to call `$util.escapeJavaScript()` function in the mapping template, as shown as follows:

```
{
  "name" : "$input.params('name')",
  "body" : $util.escapeJavaScript($input.json('$.mykey'))
}
```

Param Mapping Template Example

The following parameter-mapping example passes all 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
    }
}
```

In effect, this mapping template outputs all the request parameters in the payload as outlined as follows:

```
{
    "parameters" : {
        "path" : {
            "path_name" : "path_value",
            ...
        }
        "header" : {
            "header_name" : "header_value",
            ...
        }
        "querystring" : {
            "querystring_name" : "querystring_value",
            ...
        }
    }
}
```

Example Request and Response

Here's an example that uses all three functions:

Request Template:

```
Resource: /things/{id}

With input template:
{
    "id" : "$input.params('id')",
    "count" : "$input.path('$things').size()",
    "things" : $util.escapeJavaScript($input.json('$things'))
}

POST /things/abc
{
    "things" : {
        "1" : {},
        "2" : {},
        "3" : {}
    }
}
```

```
}
```

Response:

```
{
  "id": "abc",
  "count": "3",
  "things": {
    "1": {},
    "2": {},
    "3": {}
  }
}
```

For more mapping examples, see [Create Models and Mapping Templates for Request and Response Mappings \(p. 164\)](#)

Accessing the \$stageVariables Variable

The syntax for inserting a stage variable looks like this: `$stageVariables`.

\$stageVariables Reference

Syntax	Description
<code>\$stageVariables.<variable_name></code>	<code><variable_name></code> represents a stage variable name.
<code>\$stageVariables['<variable_name>']</code>	<code><variable_name></code> represents any stage variable name.
<code> \${stageVariables['<variable_name>']}</code>	<code><variable_name></code> represents any stage variable name.

Accessing the \$util Variable

The `$util` variable contains utility functions for use in mapping templates.

Note

Unless otherwise specified, the default character set is UTF-8.

\$util Variable Reference

Function	Description
<code>\$util.escapeJavaScript()</code>	<p>Escapes the characters in a string using JavaScript string rules.</p> <p>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:</p>

Function	Description
	<pre>\$util.escapeJavaScript(\$data).replaceAll("\\\\'","'")</pre>
<code>\$util.parseJson()</code>	<p>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:</p> <pre>{"errorMessage": {"key1": "var1", "key2": {"arr": [1, 2, 3]}}}</pre> <p>and use the following mapping template</p> <pre>#set (\$errorMessageObj = \$util.parseJson(\$input.path('\$.errorMessage'))) { "errorMessageObjKey2ArrVal" : \$errorMessageObj.key2.arr[0] }</pre> <p>You will get the following output:</p> <pre>{ "errorMessageObjKey2ArrVal" : 1 }</pre>
<code>\$util.urlEncode()</code>	Converts a string into "application/x-www-form-urlencoded" format.
<code>\$util.urlDecode()</code>	Decodes an "application/x-www-form-urlencoded" string.
<code>\$util.base64Encode()</code>	Encodes the data into a base64-encoded string.
<code>\$util.base64Decode()</code>	Decodes the data from a base64-encoded string.

Support Binary Payloads in API Gateway

In API Gateway, the API request and response can have a text or binary payload. A text payload is a UTF-8-encoded JSON string, and 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.

By default, API Gateway treats the message body as a text payload and applies any preconfigured mapping template to transform the JSON string. If no mapping template is specified, API Gateway can pass the text payload through to or from the integration endpoint without modification, provided that the passthrough behavior is enabled on the API method. For a binary payload, API Gateway simply passes through the message as-is.

For API Gateway to pass binary payloads, you add the media types to the `binaryMediaTypes` list of the `RestApi` resource or 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.
- Set the `contentHandling` property of the [Integration](#) resource to `CONVERT_TO_BINARY` to have the request payload converted from a Base64-encoded string to its binary blob, or set the property to `CONVERT_TO_TEXT` to have the request payload converted from a binary blob to a Base64-encoded string. If this property is not defined, API Gateway passes the payload through without modification. This occurs when the `Content-Type` header value matches one of the `binaryMediaTypes` entries and the [passthrough behaviors \(p. 190\)](#) are also enabled for the API.
- Set the `contentHandling` property of the [IntegrationResponse](#) resource to `CONVERT_TO_BINARY` to have the response payload converted from a Base64-encoded string to its binary blob, or set the property to `CONVERT_TO_TEXT` to have the response payload converted from a binary blob to a Base64-encoded string. If `contentHandling` is not defined, and if the `Content-Type` header of the response and the `Accept` header of the original request match an entry of the `binaryMediaTypes` list, API Gateway passes through the body. This occurs when the `Content-Type` header and the `Accept` header are the same; otherwise, API Gateway converts the response body to the type specified in the `Accept` header.

Topics

- [Content Type Conversions in API Gateway \(p. 200\)](#)
- [Enable Binary Support Using the API Gateway Console \(p. 202\)](#)
- [Enable Binary Support Using API Gateway REST API \(p. 206\)](#)
- [Import and Export Content Encodings \(p. 210\)](#)
- [Examples of Binary Support \(p. 210\)](#)

Content Type Conversions in API Gateway

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	<code>binaryMediaType</code>	<code>contentHandling</code>	Integration request payload
Text data	Any data type	Undefined	Undefined	UTF8-encoded string
Text data	Any data type	Undefined	<code>CONVERT_TO_BINARY</code>	Base64-decoded binary blob
Text data	Any data type	Undefined	<code>CONVERT_TO_TEXT</code>	UTF8-encoded string
Text data	A text data type	Set with matching media types	Undefined	Text data

Method request payload	Request Content-Type header	<code>binaryMediaType</code>	<code>contentHandling</code>	Integration request payload
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.

API Gateway Response Content Type Conversions

Integration response payload	Request Accept header	<code>binaryMediaType</code>	<code>contentHandling</code>	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
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

Integration response payload	Request Accept header	binaryMediaType	contentHandling	Method response payload
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
Binary data	A binary type	Set with matching media types	CONVERT_TO_TEXT	Base64-encoded string

Tip

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 `` 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.

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 indicating an API configuration error. You do not provide a mapping template for such a conversion, although you must enable the [passthrough behaviors \(p. 190\)](#) 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 \(p. 190\)](#) on the API.

Enable 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 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 [Create an API as an Amazon S3 Proxy \(p. 526\)](#).

To enable binary support 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 **Settings**.
- c. In the **Settings** pane, choose **Add Binary Media Type** in the **Binary Media Types** section.
- d. Type a required media type, for example, `image/png`, in the input text field. If needed, repeat this step to add more media types.
- e. Choose **Save Changes**.

Settings

Configure settings for your API deployments

API Key Source

Choose the source of your API Keys from incoming requests. Configure deployments to receive API keys from the `XAPIKEY_HEADER` or from a Custom Authorizer

API Key Source 

Content Encoding

Allow compression of response bodies based on client's Accept-Encoding header. Compression can be customized to be enabled above a certain threshold response body size. The following compression types are supported: gzip, deflate, and identity.

Content Encoding enabled

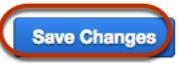
Binary Media Types

You can configure binary support for your API by specifying which media types should be treated as binary types. API Gateway will look at the **Content-Type** and **Accept** HTTP headers to decide how to handle the body.





 [Add Binary Media Type](#)

 [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. In **Content Handling**, choose an option.

/{{folder}}/{{item}} - GET - Setup

Choose the integration point for your new method.

Integration type Lambda Function i

HTTP i

Mock i

AWS Service i

AWS Region

AWS Service

AWS Subdomain

HTTP method

Action Type Use action name

Use path override

Path override (optional)

Execution role

i

Content Handling

Save

Choose **Passthrough** if you do not want to convert the body when the client and backend accepts the same binary format. Choose **Convert to text (if needed)** 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 (if needed)** 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. 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 run time.

▼ HTTP Headers

Name	Mapped from <small>i</small>	Caching	
Accept	method.request.header.Accept	<input type="checkbox"/>	 
Content-Type	method.request.header.Content-Type	<input type="checkbox"/>	 

 [Add header](#)

- e. Enable the passthrough behavior on the request body.

▼ Body Mapping Templates

Request body passthrough When no template matches the request Content-Type header i

When there are no templates defined (recommended) i

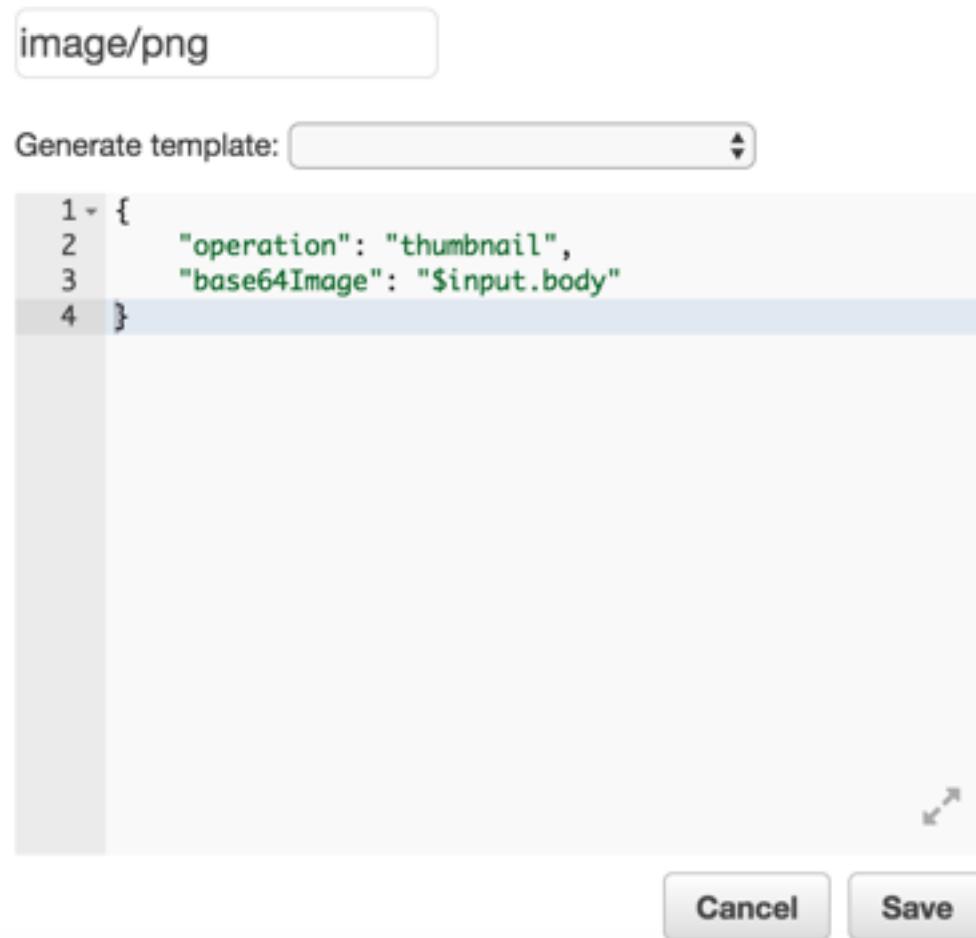
Never i

Content-Type

image/png

 [Add mapping template](#)

- f. For conversion to text, define a mapping template to put the Base64-encoded binary data into the required format.



The format of this mapping template depends on the endpoint requirements of the input.

Enable Binary Support Using 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 \(p. 206\)](#)
- [Configure Request Payload Conversions \(p. 207\)](#)
- [Configure Response Payload Conversions \(p. 207\)](#)
- [Convert Binary Data to Text Data \(p. 208\)](#)
- [Convert Text Data to a Binary Payload \(p. 208\)](#)
- [Pass through a Binary Payload \(p. 209\)](#)

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 shown next:

```
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` by submitting a `PATCH` request, as shown next:

```
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-octet-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-octet-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-octet-stream",
  }]
}
```

```
}
```

- Set the `contentHandling` property on the `Integration` resource to `CONVERT_TO_BINARY`. Do not define a mapping template. When you do not 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:

- 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-octet-stream"
  }
]
```

- 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 Swagger definition file. The extension is also used to export the API settings.

- [x-amazon-apigateway-binary-media-types Property \(p. 492\)](#)

To import and export the `contentHandling` property value on an `Integration` or `IntegrationResponse` resource, use the following API Gateway extensions to the Swagger definitions:

- [x-amazon-apigateway-integration Object \(p. 496\)](#)
- [x-amazon-apigateway-integration.response Object \(p. 502\)](#)

Examples of Binary Support

The following example demonstrates how to access a binary file in Amazon S3 or AWS Lambda through an API Gateway API. The sample API is presented in a Swagger file. The code example uses the API Gateway REST API calls.

Topics

- [Access Binary Files in Amazon S3 through an API Gateway API \(p. 210\)](#)
- [Access Binary Files in Lambda Using an API Gateway API \(p. 213\)](#)

Access Binary Files in Amazon S3 through an API Gateway API

The following examples show the Swagger 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

- [Swagger File of a Sample API to Access Images in Amazon S3 \(p. 210\)](#)
- [Download an Image from Amazon S3 \(p. 212\)](#)
- [Upload an Image to Amazon S3 \(p. 213\)](#)

Swagger File of a Sample API to Access Images in Amazon S3

The following Swagger 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.

```
{  
  "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": {}  
}
```

```

"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 like this, as shown in the bold-faced Swagger definition block below:

```

    "x-amazon-apigateway-integration": {
        "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
        "responses": {
            "default": {
                "statusCode": "500"
            },
            "2\\d{2)": {
                "statusCode": "200",
                "responseTemplates": {
                    "application/json": "{'image': 'data:image/jpeg;base64," + $input.body}"
                }
            }
        }
    }
}

```

```
        "responseTemplates": {  
            "application/json": "{\n                \"image\": \"$input.body\"\n            }\n        },
```

The request to download the image file looks like this:

```
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 this:

```
200 OK HTTP/1.1  
{  
    "image": "W3JhdBieXRlc10="  
}
```

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 this:

```
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  
  
W3JhdBieXRlc10=
```

Notice that 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 this:

```
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 a Swagger file. The code example uses the API Gateway REST API calls.

Topics

- [Swagger File of a Sample API to Access Images in Lambda \(p. 214\)](#)
- [Download an Image from Lambda \(p. 215\)](#)
- [Upload an Image to Lambda \(p. 216\)](#)

Swagger File of a Sample API to Access Images in Lambda

The following Swagger file shows an example API that illustrates downloading an image file from Lambda and uploading an image file to Lambda.

```
{  
  "swagger": "2.0",  
  "info": {  
    "version": "2016-10-21T17:26:28Z",  
    "title": "ApiName"  
  },  
  "host": "abcdefghijklmnopqrstuvwxyz.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 this:

```
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 this:

```
200 OK HTTP/1.1
{
  "image": "W3JhdBieXRlc10="
}
```

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 this:

```
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
W3JhdBieXRlc10=
```

The successful response looks like this:

```
200 OK
```

Enable Payload Compression for an API

API Gateway allows your client to call your API with compressed payloads using one of the [supported content codings \(p. 219\)](#). 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 `minimumCompressionSize` 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 the method request payload 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 Compression for an API \(p. 217\)](#)
- [Call an API Method with a Compressed Payload \(p. 220\)](#)
- [Receive an API Response with a Compressed Payload \(p. 220\)](#)

Enable Compression for an API

You can enable compression for an API using the API Gateway console, AWS CLI/SDK, or API Gateway REST API. The steps are detailed in the following sections.

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.

Topics

- [Enable Compression for an API Using the API Gateway Console \(p. 217\)](#)
- [Enable Compression for an API Using AWS CLI \(p. 218\)](#)
- [Enable Compression for an API Using the API Gateway REST API \(p. 218\)](#)
- [Content Codings Supported by API Gateway \(p. 219\)](#)

Enable 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.
2. Choose an existing API or create a new one.
3. In the primary navigation pane, choose **Settings** under the API you chose or the one you created.
4. Under the **Content Encoding** section in the **Settings** pane, select the **Content Encoding enabled** option to enable payload compression. Type a number for the minimum compression size (in bytes) next to **Minimum body size required for compression**. To disable the compression, clear the **Content Encoding enabled** option.
5. Choose **Save Changes**.

Enable 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" \
--minimumCompressionSize 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.

Enable Compression for an API Using the API Gateway REST API

To use the [API Gateway REST API](#) to enable compression on a new API, call `restapi:create` as follows:

```
POST /restapis
Host: apigateway.{region}.amazonaws.com
Authorization: apigateway.{region}.amazonaws.com
Content-Type: application/json
Content-Length: ...

{
    "name" : "My test API",
    "minimumCompressionSize": 0
```

```
}
```

To use the [API Gateway REST API](#) to enable compression on an existing API, call `restapi:update` as follows:

```
PATCH /restapis/{restapi_id}
Host: apigateway.{region}.amazonaws.com
Authorization: ...
Content-Type: application/json
Content-Length: ...

{
  "patchOperations" : [ {
    "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 disable compression by using the API Gateway REST API, call `restapi:update` as follows:

```
PATCH /restapis/{restapi_id}
Host: apigateway.{region}.amazonaws.com
Authorization: ...
Content-Type: application/json
Content-Length: ...

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/minimumCompressionSize"
  } ]
}
```

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=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 \(p. 219\)](#).

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 be 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 \(p. 219\)](#).

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 will 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.

Enable 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.

Topics

- [Overview of Basic Request Validation in API Gateway \(p. 222\)](#)
- [Set up Basic Request Validation in API Gateway \(p. 222\)](#)
- [Test Basic Request Validation in API Gateway \(p. 227\)](#)
- [Swagger Definitions of a Sample API with Basic Request Validation \(p. 230\)](#)

Overview of Basic Request Validation in API Gateway

API Gateway can perform the basic validation. This enables you, the API developer, to focus on app-specific deep validation in the backend. For the basic 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 non-blank.
- The applicable request payload adheres to the configured [JSON schema request model \(p. 170\)](#) of the method.

To enable basic 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 [request body passthrough \(p. 190\)](#) are two separate issues. When a request payload cannot be validated because no model schema can be matched, you can choose to passthrough or block the original payload. For example, when you enable request validation with a mapping template for the `application/json` media type, you may want to pass an XML payload through to the backend even though the enabled request validation will fail. This may be the case if you expect to support the XML payload on the method in the future. To fail the request with an XML payload, you must explicitly choose the `NEVER` option for the content passthrough behavior.

Set up Basic Request Validation in API Gateway

You can set up request validators in an API's Swagger definition file and then import the Swagger definitions into API Gateway. You can also set them up in the API Gateway console or by calling the API Gateway REST API, AWS CLI, or one of the AWS SDKs. Here, we show how to do this with a Swagger file, in the console, and using the API Gateway REST API.

Topics

- [Set up Basic Request Validation by Importing API Swagger Definition \(p. 222\)](#)
- [Set up Request Validators Using the API Gateway REST API \(p. 225\)](#)
- [Set up Basic Request Validation Using the API Gateway Console \(p. 226\)](#)

Set up Basic Request Validation by Importing API Swagger Definition

The following steps describe how to enable basic request validation by importing a Swagger file.

To enable request validation by importing a Swagger file into API Gateway

1. Declare request validators in Swagger by specifying a set of the [x-amazon-apigateway-request-validator Object \(p. 506\)](#) objects in the [x-amazon-apigateway-request-validation Object \(p. 505\)](#) map at the API level. For example, the [sample API Swagger file \(p. 230\)](#) contains the `x-amazon-apigateway-request-validation` map, with the validators' names as the keys.

```
{  
    "swagger": "2.0",  
    "info": {  
        "title": "ReqValidation Sample",  
        "version": "1.0.0"  
    },  
    "schemes": [  
        "https"  
    ],  
    "basePath": "/v1",  
    "produces": [  
        "application/json"  
    ],  
    "x-amazon-apigateway-request-validation": {  
        "all": {  
            "validateRequestBody": true,  
            "validateRequestParameters": true  
        },  
        "params-only": {  
            "validateRequestBody": false,  
            "validateRequestParameters": true  
        }  
    },  
    ...  
}
```

You select a validator's name when enabling the validator on the API or on a method, as shown in the next step.

2. To enable a request validator on all methods of an API, specify an [x-amazon-apigateway-request-validator Property \(p. 504\)](#) property at the API level of the API Swagger definition file. To enable a request validator on an individual method, specify the `x-amazon-apigateway-request-validator` property at the method level. For example, the following `x-amazon-apigateway-request-validator` property enables the `params-only` validator on all API methods, unless otherwise overridden.

```
{  
    "swagger": "2.0",  
    "info": {  
        "title": "ReqValidation Sample",  
        "version": "1.0.0"  
    },  
    "schemes": [  
        "https"  
    ],  
    "basePath": "/v1",  
    "produces": [  
        "application/json"  
    ],  
    ...  
    "x-amazon-apigateway-request-validator": "params-only",  
    ...  
}
```

To enable a request validator on an individual method, specify the `x-amazon-apigateway-request-validator` property at the method level. For example, the following `x-amazon-apigateway-request-validator` property enables the `all` validator on the `POST /validation` method. This overrides the `params-only` validator that is inherited from the API.

```
{  
    "swagger": "2.0",  
    "info": {  
        "title": "ReqValidation Sample",  
        "version": "1.0.0"  
    },  
    "schemes": [  
        "https"  
    ],  
    "basePath": "/v1",  
    "produces": [  
        "application/json"  
    ],  
    ...  
    "paths": {  
        "/validation": {  
            "post": {  
                "x-amazon-apigateway-request-validator" : "all",  
                ...  
            },  
            ...  
        }  
    }  
}
```

3. In API Gateway, create the API with request validators enabled by importing this [Sample API Swagger Definition \(p. 230\)](#):

```
POST /restapis?mode=import&failonwarning=true HTTP/1.1  
Content-Type: application/json  
Host: apigateway.us-east-1.amazonaws.com  
X-Amz-Date: 20170306T234936Z  
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170306/us-east-1/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature={sig4_hash}
```

Copy the JSON object from this API Swagger Definition (p. 230) and paste it here.

4. Deploy the newly created API (`fjd6crafxc`) to a specified stage (`testStage`).

```
POST /restapis/fjd6crafxc/deployments HTTP/1.1  
Content-Type: application/json  
Host: apigateway.us-east-1.amazonaws.com  
X-Amz-Date: 20170306T234936Z  
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170306/us-east-1/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature={sig4_hash}  
  
{  
    "stageName" : "testStage",  
    "stageDescription" : "Test stage",  
    "description" : "First deployment",  
    "cacheClusterEnabled" : "false"  
}
```

For instructions on how to test the request validation using the API Gateway REST API, see [Test Basic Request Validation Using the API Gateway REST API \(p. 227\)](#). For instructions on how to test using the API Gateway console, see [Test Basic Request Validation Using the API Gateway Console \(p. 229\)](#).

Set up Request Validators Using the API Gateway REST API

In the API Gateway REST API, a request validator is represented by a [RequestValidator](#) resource. To have an API support the same request validators as the [Sample API \(p. 230\)](#), add to the [RequestValidators](#) collection a parameters-only validator with `params-only` as the key, and add a full validator with `all` as its key.

To enable the basic request validation using the API Gateway REST API

We assume that you have an API similar to the [sample API \(p. 230\)](#), but have not set up the request validators. If your API already has request validators enabled, call the appropriate `requestvalidator:update` or `method:put` action instead of `requestvalidator:create` or `method:put`.

1. To set up the `params-only` request validator, call the `requestvalidator:create` action as follows:

```
POST /restapis/restapi-id/requestvalidators HTTP/1.1
Content-Type: application/json
Host: apigateway.region.amazonaws.com
X-Amz-Date: 20170223T172652Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170223/region/apigateway/
aws4_request, SignedHeaders=content-type;host;x-amz-date, Signature={sig4_hash}

{
  "name" : "params-only",
  "validateRequestBody" : "false",
  "validateRequestParameters" : "true"
}
```

2. To set up the `all` request validator, call the `requestvalidator:create` action as follows:

```
POST /restapis/restapi-id/requestvalidators HTTP/1.1
Content-Type: application/json
Host: apigateway.region.amazonaws.com
X-Amz-Date: 20170223T172652Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170223/region/apigateway/
aws4_request, SignedHeaders=content-type;host;x-amz-date, Signature={sig4_hash}

{
  "name" : "all",
  "validateRequestBody" : "true",
  "validateRequestParameters" : "true"
}
```

If the preceding validator keys already exist in the `RequestValidators` map, call the `requestvalidator:update` action instead to reset the validation rules.

3. To apply the `all` request validator to the `POST` method, call `method:put` to enable the specified validator (as identified by the `requestValidatorId` property) or call `method:update` to update the enabled validator.

```
PUT /restapis/restapi-id/resources/resource-id/methods/POST HTTP/1.1
Content-Type: application/json
Host: apigateway.region.amazonaws.com
X-Amz-Date: 20170223T172652Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170223/region/apigateway/
aws4_request, SignedHeaders=content-type;host;x-amz-date, Signature={sig4_hash}
```

```
{  
    "authorizationType" : "NONE",  
    ...  
    "requestValidatorId" : "all"  
}
```

Set up Basic Request Validation Using the API Gateway Console

The API Gateway console lets you set up the basic request validation on a method using one of the three validators:

- **Validate body:** This is the body-only validator.
- **Validate query string parameters and headers:** This is the parameters-only validator.
- **Validate body, query string parameters, and headers:** This validator is for both body and parameters validation.

When you choose one of the above validators to enable it on an API method, the API Gateway console will add the validator to the API's [RequestValidators](#) map, if the validator has not already been added to the validators map of the API.

To enable a request validator on a method

1. Sign in to the API Gateway console, if not already logged in.
2. Create a new or choose an existing API.
3. Create a new or choose an existing resource of the API.
4. Create a new or choose an existing method the resource.
5. Choose **Method Request**.
6. Choose the pencil icon of **Request Validator** under **Settings**.
7. Choose **Validate body**, **Validate query string parameters and headers** or **Validate body, query string parameters, and headers** from the **Request Validator** drop-down list and then choose the check mark icon to save your choice.

◀ Method Execution / - GET - Method Request



Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization **NONE**

Request Validator

API Key Required **false**

To test and use the request validator in the console, follow the instructions in [Test Basic Request Validation Using the API Gateway Console \(p. 229\)](#).

Test Basic Request Validation in API Gateway

Choose one of the following topics for instructions on testing the basic request validation against the [sample API \(p. 230\)](#).

Topics

- [Test Basic Request Validation Using the API Gateway REST API \(p. 227\)](#)
- [Test Basic Request Validation Using the API Gateway Console \(p. 229\)](#)

Test Basic Request Validation Using the API Gateway REST API

To see the invocation URL of the deployed API, you can export the API from the stage, making sure to include the `Accept: application/json` or `Accept: application/yaml` header:

```
GET /restapis/fjd6crafxc/stages/testStage/exports/swagger?extensions=validators HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: apigateway.us-east-1.amazonaws.com
X-Amz-Date: 20170306T234936Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20170306/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature={sig4_hash}
```

You can ignore the `?extensions=validators` query parameter, if you do not want to download the Swagger specifications related to the request validation.

To test request validation using the API Gateway REST API calls

1. Call `GET /validation?q1=cat`.

```
GET /testStage/validation?q1=cat HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

Because the required parameter of `q1` is set and not blank, the request passes the validation. API Gateway returns the following 200 OK response:

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

2. Call `GET /validation`.

```
GET /testStage/validation HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

Because the required parameter of `q1` is not set, the request fails to pass the validation. API Gateway returns the following 400 Bad Request response:

```
{ "message": "Missing required request parameters: [q1]" }
```

3. Call GET `/validation?q1=.`.

```
GET /testStage/validation?q1= HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

Because the required parameter of `q1` is blank, the request fails to pass the validation. API Gateway returns the same 400 Bad Request response as in the previous example.

4. Call POST `/validation`.

```
POST /testStage/validation HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
h1: v1

{
    "name" : "Marco",
    "type" : "dog",
    "price" : 260
}
```

Because the required header parameter of `h1` is set and not blank and the payload format adheres to the `requestDataModel` required properties and associated constraints, the request passes the validation. API Gateway returns the following successful response.

```
{ "pet": {
        "name": "Marco",
        "type": "dog",
        "price": 260
    },
    "message": "success"
}
```

5. Call POST `/validation`, without specifying the `h1` header or setting its value blank.

```
POST /testStage/validation HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json

{
    "name" : "Marco",
    "type" : "dog",
```

```
    "price" : 260
}
```

Because the required header parameter of h1 is missing or set to blank, the request fails to pass the validation. API Gateway returns the following 400 Bad Request response:

```
{
  "message": "Missing required request parameters: [h1]"
}
```

6. Call POST /validation, setting the type property of the payload to bird.

```
POST /testStage/validation HTTP/1.1
Host: fjd6crafxc.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
X-Amz-Date: 20170309T000215Z
h1: v1

{
  "name" : "Molly",
  "type" : "bird",
  "price" : 269
}
```

Because the type property value is not a member of the enumeration ["dog", "cat", "fish"], the request fails to pass the validation. API Gateway returns the following 400 Bad Request response:

```
{
  "message": "Invalid request body"
}
```

Setting price to 501 violates the constraint on the property. This causes the validation to fail and the same 400 Bad Request response is returned.

Test Basic Request Validation Using the API Gateway Console

The following steps describe how to test basic request validation in the API Gateway console.

To test the request validation on a method using TestInvoke in the API Gateway console

While signed in to the API Gateway console, do the following:

1. Choose **Resources** for the API for which you have configured a request validators map.
2. Choose a method for which you have enabled the request validation with a specified request validator.
3. Under **Method Execution**, in the **Client** box, choose **Test**.
4. Try different values for required request parameter or applicable body, and then choose **Test** to see the response.

When the method call passes validation, you should get expected responses. If validation fails, the following error message returns if the payload is not the correct format:

```
{
```

```
    "message": "Invalid request body"
}
```

If the request parameters are not valid, the following error message returns:

```
{
    "message": "Missing required request parameters: [p1]"
}
```

Swagger Definitions of a Sample API with Basic Request Validation

The following Swagger 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-validation`s 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` schema. 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.

For more information about the behavior of this API, see [Enable Request Validation in API Gateway \(p. 221\)](#).

```
{
    "swagger": "2.0",
    "info": {
        "title": "ReqValidators Sample",
        "version": "1.0.0"
    },
    "schemes": [
        "https"
    ],
    "basePath": "/v1",
    "produces": [
        "application/json"
    ],
    "x-amazon-apigateway-request-validation": {
        "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": [
                    "q1"
                ]
            }
        }
    }
}
```

```
{  
    "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": {
      }
    }
  }
}
```

}

Update and Maintain an API in Amazon API Gateway

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.
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.

Topics

- [Update an API Endpoint Type in API Gateway \(p. 234\)](#)
- [Maintain an API Using the API Gateway Console \(p. 236\)](#)

Update an API Endpoint Type in API Gateway

Updating an API endpoint type requires you to update the API's configuration. You can update an existing API type from edge-optimized to regional, and vice-versa, using the API Gateway console, AWS CLI, an AWS SDK for API Gateway, or the API Gateway REST API. The update operation may take up to 60 seconds to complete.

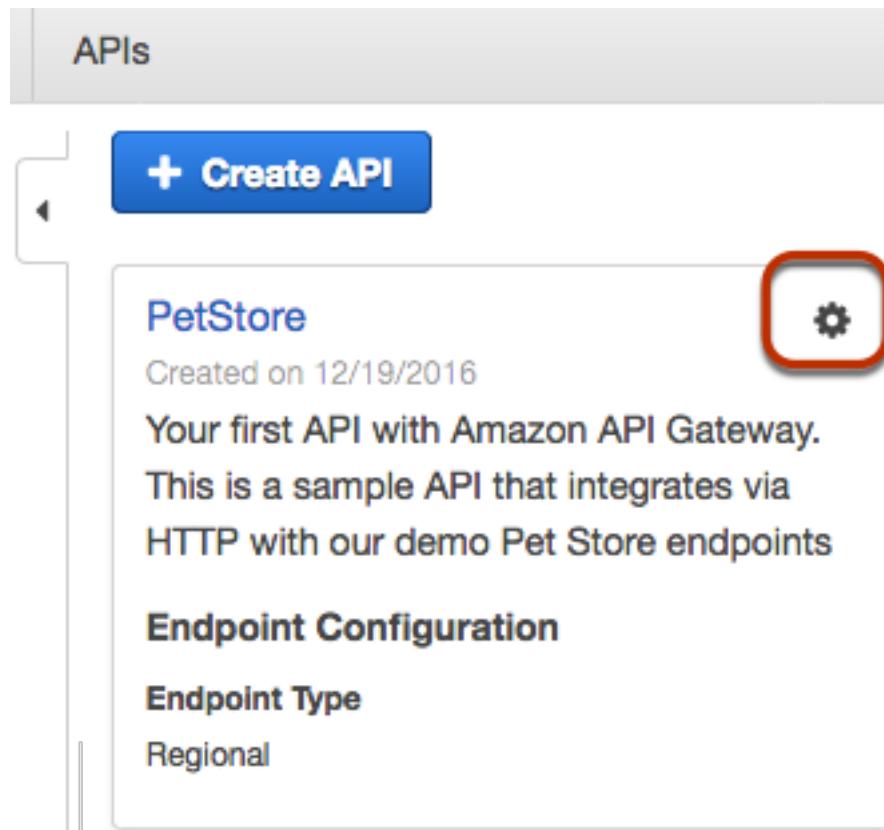
An edge-optimized API may have different behaviors than a regional API. For example, an edge-optimized API removes the Content-MD5 header. Any MD5 hash value passed to the backend can be expressed in a request string parameter or a body property. However, the regional API passes this header through, 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 Update an API Endpoint \(p. 234\)](#)
- [Use the AWS CLI to Update an API Endpoint \(p. 235\)](#)
- [Use the API Gateway REST API to Update an API Endpoint \(p. 236\)](#)

Use the API Gateway Console to Update an API Endpoint

1. Sign in to the console and choose **APIs** in the primary navigation pane.
2. Choose the settings (gear icon) of an API under **+ Create API**.



3. Change the **Endpoint Type** option under **Endpoint Configuration** from Edge Optimized to Regional or from Regional to Edge Optimized.
4. Choose **Save** to start the update.

Use the AWS CLI to Update an API Endpoint

To use the AWS CLI commands to update an edge-optimized API of {api-id}, call the [restapi:update](#) 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.

Use the API Gateway REST API to Update an API Endpoint

To use the API Gateway REST API to update an edge-optimized API of `{api-id}`, call the [restapi:update](#) as follows:

```
PATCH /restapis/{api-id}

{
  "patchOperations" : [ {
    "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, to update a regional API to an edge-optimized API, call the [restapi:update](#) as follows:

```
PATCH /restapis/{api-id}

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/endpointConfiguration/types/REGIONAL",
    "value" : "EDGE"
  }]
}
```

Because [restapi:put](#) is for updating API definitions, it is not applicable to updating an API endpoint type.

Maintain an API Using the API Gateway Console

Topics

- [View a List of APIs in API Gateway \(p. 237\)](#)
- [Delete an API in API Gateway \(p. 237\)](#)
- [Delete a Resource in API Gateway \(p. 237\)](#)
- [View a Methods List in API Gateway \(p. 237\)](#)
- [Delete a Method in API Gateway \(p. 238\)](#)

View a List of APIs in API Gateway

Use the API Gateway console to view a list of APIs.

View a List of APIs with the API Gateway Console

You must have an API available in API Gateway. Follow the instructions in [Creating an API in Amazon API Gateway \(p. 81\)](#).

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. The list of APIs is displayed.

Delete an API in API Gateway

Use the API Gateway console to delete an API.

Warning

Deleting an API means that you can no longer call it. This action cannot be undone.

Delete an API with the API Gateway Console

You must have deployed the API at least once. Follow the instructions in [Deploying an API in Amazon API Gateway \(p. 370\)](#).

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API you want to delete, choose **Resources**.
3. Choose **Delete API**.
4. When prompted to delete the API, choose **Ok**.

Delete a Resource in API Gateway

Use the API Gateway console to delete a resource.

Warning

When you delete a resource, you also delete its child resources and methods. Deleting a resource may cause part of the corresponding API to be unusable. Deleting a resource cannot be undone.

Delete a Resource with the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API for the resource you want to delete, choose **Resources**.
3. In the **Resources** pane, choose the resource, and then choose **Delete Resource**.
4. When prompted, choose **Delete**.

View a Methods List in API Gateway

Use the API Gateway console to view a list of methods for a resource.

View a Methods List with the API Gateway Console

You must have methods available in API Gateway. Follow the instructions in [Build an API with HTTP Custom Integration \(p. 44\)](#).

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

2. In the box that contains the name of the API, choose **Resources**.
3. The list of methods is displayed in the **Resources** pane.

Tip

You may need to choose the arrow next to one or more resources to display all of the available methods.

Delete a Method in API Gateway

Use the API Gateway console to delete a method.

Warning

Deleting a method may cause part of the corresponding API to become unusable. Deleting a method cannot be undone.

Delete a Method with the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API for the method, choose **Resources**.
3. In the **Resources** pane, choose the arrow next to the resource for the method.
4. Choose the method, and then choose **Delete Method**.
5. When prompted, choose **Delete**.

Import an API into API Gateway

You can use the API Gateway Import API feature to import an API from an external definition file into API Gateway. Currently, the Import API feature supports [Swagger v2.0](#) definition files.

With the Import API, you can either create a new API by submitting a [POST request](#) that includes a Swagger definition in the payload and endpoint configuration, or you can update an existing API by using a [PUT request](#) that contains a Swagger definition in the payload. You can update an API by overwriting it with a new definition, or merge a definition with an existing API. You specify the options using a `mode` query parameter in the request URL .

Note

For RAML API definitions, you can continue to use [API Gateway Importer](#).

Besides making explicit calls to the REST API, as described below, you can also use the Import API feature in the API Gateway console. For a quick start to using the Import API feature from the API Gateway console, see [Build an API Gateway API from an Example \(p. 9\)](#).

Topics

- [Import an Edge-Optimized API into API Gateway \(p. 238\)](#)
- [Import a Regional API into API Gateway \(p. 240\)](#)
- [Import a Swagger File to Update an Existing API Definition \(p. 240\)](#)
- [Set the Swagger basePath Property \(p. 242\)](#)
- [Errors and Warnings during Import \(p. 243\)](#)

Import an Edge-Optimized API into API Gateway

You can import an API Swagger definition file to create a new edge-optimized API by specifying the `EDGE` endpoint type as an additional input, besides the Swagger file, to the import operation. You can do so using the API Gateway console, AWS CLI, an AWS SDK, or the API Gateway REST API.

Topics

- [Import an Edge-Optimized API Using the API Gateway Console \(p. 239\)](#)
- [Import an Edge-Optimized API Using the AWS CLI \(p. 239\)](#)
- [Import an Edge-Optimized API Using the API Gateway REST API \(p. 239\)](#)

Import an Edge-Optimized API Using the API Gateway Console

To import an API of an edge-optimized API endpoint type using the API Gateway console, do the following:

1. Sign in to the API Gateway console and choose **+ Create API**.
2. Select the **Import from Swagger** option under **Create new API**.
3. Copy an API Swagger definition and paste it into the code editor, or choose **Select Swagger File** to load a Swagger file from a local drive.
4. Under **Settings**, for **Endpoint Type**, choose **Edge optimized..**.
5. Choose **Import** to start importing the Swagger definitions.

Import an Edge-Optimized API Using the AWS CLI

To import an API from a Swagger 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_Swagger_template.json'
```

or with an explicit specification of the `endpointConfigurationTypes` query string parameter to `EDGE`:

```
aws apigateway import-rest-api \
  --endpointConfigurationTypes 'EDGE' \
  --fail-on-warnings \
  --body 'file://path/to/API_Swagger_template.json'
```

Import an Edge-Optimized API Using the API Gateway REST API

To use the API Gateway REST API to create a regional API by importing a Swagger definition file, call the following `restapi:import` link-relation:

```
POST /restapis?mode=import&failonwarnings=true
Host: apigateway.us-west-2.amazonaws.com
Content-Type:application/json
Content-Length: ...

{
    //API Swagger definition
}
```

or with an explicit specification of the `endpointConfigurationTypes` query string parameter to `EDGE`:

```
POST /restapis?mode=import&failonwarnings=true&endpointConfigurationTypes=EDGE
Host: apigateway.us-west-2.amazonaws.com
```

```
Content-Type:application/json
Content-Length: ...

{
    //API Swagger definition
}
```

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, AWS CLI, an AWS SDK, or the API Gateway REST API.

When you export an API, the API endpoint configuration is not included in the exported API definitions.

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 and choose **+ Create API**.
2. Select the **Import from Swagger** option under **Create new API**.
3. Copy an API Swagger definition and paste it into the code editor, or choose **Select Swagger File** to load a Swagger file from a local drive.
4. Under **Settings**, for **Endpoint Type**, choose **Regional**.
5. Choose **Import** to start importing the Swagger definitions.

Import a Regional API Using the AWS CLI

To import an API from a Swagger definition file using the AWS CLI, use the `import-rest-api` command:

```
aws apigateway import-rest-api \
--endpointConfigurationTypes 'REGIONAL' \
--fail-on-warnings \
--body 'file://path/to/API_Swagger_template.json'
```

Import a Regional API Using the API Gateway REST API

To use the API Gateway REST API to create a regional API by importing a Swagger definition file, call the following `restapi:import` link-relation:

```
POST /restapis?mode=import&failonwarnings=true&endpointConfigurationTypes=REGIONAL
Host: apigateway.us-west-2.amazonaws.com
Content-Type:application/json
Content-Length: ...

{
    //API Swagger definition
}
```

Import a Swagger 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 will yield the following methods:

```
GET /a
```

which will return 200 and 400 responses, and

```
POST /a
```

which will return 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 are not specifically defined in the imported definition will be 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 are not specifically defined in the imported definition will be 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 a Swagger 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: ...
```

A Swagger API definition in JSON (p. 521)

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 Swagger definition:

```
PUT /restapis/<restapi_id>?mode=overwrite
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
Content-Length: ...
```

A Swagger API definition in JSON (p. 521)

When the `mode` query parameter is not 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 will put the API into the same final state as if the first operation had succeeded.

Set the Swagger basePath Property

In Swagger, 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 three options for interpreting the `basePath` property during an import:

ignore

If the Swagger 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
```

will result 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 does not include a *Base Path* and a *Stage* value that refers to your production stage.

Note

API Gateway will automatically create a root resource for you, even if it is not explicitly declared in your definition file.

When unspecified, `basePath` takes `ignore` by default.

prepend

If the Swagger 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
```

will result 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 will automatically create intermediate resources for you, even if they are not explicitly declared in your definition.

split

If the Swagger 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
```

will result 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.

Errors and Warnings during Import

Errors during Import

During the import, errors can be generated for major issues like an invalid Swagger document. Errors are returned as exceptions (e.g., `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.

Controlling Access to an API in API Gateway

API Gateway supports multiple mechanisms of access control, including metering or tracking API uses by clients using API keys. The standard AWS IAM roles and policies offer flexible and robust access controls that can be applied to an entire API set or individual methods. Lambda authorizers and Amazon Cognito user pools provide customizable authorization and authentication solutions.

Topics

- [Control Access to an API with Amazon API Gateway Resource Policies \(p. 245\)](#)
- [Control Access to an API with IAM Permissions \(p. 252\)](#)
- [Enable CORS for an API Gateway Resource \(p. 267\)](#)
- [Use API Gateway Lambda Authorizers \(p. 272\)](#)
- [Use Amazon Cognito User Pools \(p. 286\)](#)
- [Use Client-Side SSL Certificates for Authentication by the Backend \(p. 293\)](#)
- [Create and Use API Gateway Usage Plans \(p. 314\)](#)

Control Access to an API with Amazon 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 user or role) can invoke the API. You can 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.

You can use resource policies for all endpoint types in API Gateway: edge-optimized and regional.

You can attach a resource policy to an API using the AWS console, AWS CLI, or AWS SDKs.

API Gateway resource policies are different from IAM policies. IAM policies are attached to IAM entities (users, groups, groups, or roles) and define what actions those entities 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 (IAM) 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 \(p. 246\)](#)
- [API Gateway Resource Policy Examples \(p. 247\)](#)
- [Create and Attach an API Gateway Resource Policy to an API \(p. 248\)](#)
- [Interactions Between API Gateway Resource Policies and IAM Policies \(p. 250\)](#)
- [AWS Conditions that can be used in API Gateway Resource Policies \(p. 251\)](#)

Access Policy Language Overview for Amazon API Gateway

The topics in this section describe 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*.

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.

For the format of the `Resource` element, see [Resource Format of Permissions for Executing API in API Gateway \(p. 260\)](#).

- **Actions** – For each resource, Amazon API Gateway supports a set of operations. You identify resource operations you will allow (or deny) by using action keywords.

For example, the `apigateway: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 \(p. 260\)](#).

- **Effect** – What the effect will be 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.
- **Principal** – The account or user who is allowed access to the actions and resources in the statement. In a resource policy, the principal is the IAM user or account who is the recipient of this permission.

The following example resource policy shows the preceding common policy elements. The policy grants access to a specific API `api-id` in the specified `region` to any user whose source IP address is in the IP range `203.0.113.0` to `203.0.113.255` only. The policy denies all access to the API if the user's source IP is not within the range.

```
{  
    "Id": "Policy1513815114229",  
    "Version": "2018-01-01",  
    "Statement": [  
        {  
            "Sid": "Stmt1513815112643",  
            "Effect": "Deny",  
            "Principal": "*",  
            "Action": [  
                "execute-api:Invoke"  
            ],  
            "Condition": {  
                "NotIpAddress": {  
                    "aws:SourceIp": [  
                        "203.0.113.0",  
                        "203.0.113.255"  
                    ]  
                }  
            },  
            "Resource": [  
                "arn:aws:execute-api:  
            ]  
        }  
    ]  
}
```

```
        "arn:aws:execute-api:region:account-id:api-id/stage/method/path"  
    }  
}  
]
```

API Gateway Resource Policy Examples

This section presents a few examples of typical use cases for API Gateway resource policies. The policies use **account-id** and **api-id** strings in the resource value. To test these policies, you need to replace these strings with your own account ID and API ID. For information about access policy language, see [Access Policy Language Overview for Amazon API Gateway \(p. 246\)](#).

Example: Allow users in another AWS account to use an API

The following example resource policy grants access to two users in an AWS account via [Signature Version 4 \(SigV4\)](#) protocols. Specifically, Alice and the root user for the same AWS account are granted the `execute-api:Invoke` action to execute the `GET` action on the `pets` resource (API).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": [  
                    "arn:aws:iam::account-id:user/Alice",  
                    "account-id"  
                ]  
            },  
            "Action": "execute-api:Invoke",  
            "Resource": [  
                "arn:aws:execute-api:region:account-id:api-id/stage/GET/pets"  
            ]  
        }  
    ]  
}
```

Example: Deny API traffic based on source IP address or range

The following example resource policy is a "blacklist" policy that denies (blocks) incoming traffic to an API from two specified source IP addresses.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "execute-api:Invoke",  
            "Resource": [  
                "arn:aws:execute-api:region:account-id:api-id/*"  
            ]  
        },  
        {  
            "Effect": "Deny",  
            "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"]
            }
        }
    ]
}
```

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.

The resource policy can be attached to the API when the API is created, or it can be updated later.

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 an API in Amazon API Gateway \(p. 370\)](#).

Access can be controlled by IAM condition elements, including conditions on AWS account or IP range. If the `Principal` in the policy is set to `"*"`, other authorization types can be used alongside the resource policy. If the `Principal` is set to `"AWS": ...`, authorization will fail 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.

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 the API name.
3. In the left-hand navigation pane, choose **Resource Policy**.
4. If desired, choose one of the **Examples**. In the example policies, placeholders are enclosed in double curly braces (`{"{}"}`). Replace each of the placeholders (including the curly braces) with the necessary information.
If you don't use one of the **Examples**, enter your resource policy.
5. Choose **Save**.

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 (API Gateway API)

To use the [API Gateway REST API](#) to create a new API and attach a resource policy to it, call the [create-rest-api](#) command as follows:

```
POST /restapis

{
  "name": "api-name",
  "policy": "{\"jsonEscapedPolicyDocument}"
}
// simplified format supported here because apiId is not known yet and partition/region/
account can be derived at import time
// "Resource": [
//   "execute-api:/stage/method/path"
// ]
```

To use the [API Gateway REST API](#) to attach a resource policy to an existing API, call the [restapi:update](#) command as follows:

```
PATCH /restapis/api-id

{
  "patchOperations" : [ {
    "op": "replace",
    "path": "/policy",
    "value": "{\"jsonEscapedPolicyDocument}"
  } ]
}
```

Swagger Example of Attaching a API Gateway Resource Policy

The [restapi:import](#) command can be used to import a Swagger definition of an API with attached resource policy, as shown in the following example:

```
{
  "swagger": "2.0",
  "x-amazon-apigateway-policy": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "AWS": [
            "arn:aws:iam::111122223333:user/Alice",
            "arn:aws:iam::111122223333:root"
          ]
        },
        "Action": "execute-api:Invoke",
        "Resource": [
          "execute-api:/stage/method/path" // simplified format supported here because apiId
is not known yet and partition/region/account can be derived at import time
        ]
      }
    ]
  }
}
```

```

        ]
    ],
    "info" : {
        "title" : "Example"
    },
    "schemes": [
        "https"
    ],
    "paths": {...}
}

```

Interactions Between API Gateway Resource Policies and IAM Policies

The following tables list the resulting behavior when access to an API Gateway API is controlled by both an IAM policy and an API Gateway resource policy.

For more 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](#).

Note

"Implicit deny" is the same thing as "Deny by default."

"Implicit deny" is different from "Explicit deny." For more information, see [The Difference Between Denying by Default and Explicit Deny](#).

Account A Calls API Owned by Account A

IAM User Policy	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
Neither Allow nor Deny	Deny	Explicit Deny
Deny	Allow	Explicit Deny
Deny	Neither Allow nor Deny	Explicit Deny
Deny	Deny	Explicit Deny

The following table shows the behavior when the IAM or resource policy is silent is that the access is denied (cross account access requires that both the resource policy and the IAM policy explicitly grant access).

Account B Calls API Owned by Account A

IAM User Policy	API Gateway Resource Policy	Resulting Behavior
Allow	Allow	Allow

IAM User Policy	API Gateway Resource Policy	Resulting Behavior
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
Deny	Deny	Explicit Deny

AWS Conditions that can be used in API Gateway Resource Policies

The following table contains the complete list of AWS condition keys that can be used in resource policies for APIs in API Gateway for each authorization type.

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:MultiFactorAuthType	Key is present only in requests that are signed using temporary security credentials.	Yes	IAM
aws:MultiFactorAuthStatus	Key is present only if MFA is present in the requests.	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

Condition Keys	Criteria	Needs AuthN?	Authorization Type
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 Amazon API Gateway 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 an [IAM user](#) representing the developer, to an [IAM group](#) containing the user, or to an [IAM role](#) assumed by the user.

In this IAM policy document, the IAM Resource element contains a list of API Gateway API entities, including [API Gateway resources](#) and [API Gateway link-relations](#). The IAM Action element contains the required API Gateway API-managing actions. These actions are declared in the apigateway:[HTTP_VERB](#) format, where apigateway designates the underlying API management component of API Gateway, and [HTTP_VERB](#) represents HTTP verbs supported by API Gateway.

For more information on how to use this permissions model, see [Control Access for Managing an API \(p. 254\)](#).

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 the IAM user authentication is enabled. The API developer sets the method's `authorizationType` property to `AWS_IAM` to require that the caller submit the IAM user's access keys to be authenticated. Then, you attach the policy to an IAM user representing the API caller, to an IAM group containing the user, or to an IAM role assumed by the user.

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 \(p. 258\)](#).

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 IAM 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 an IAM user access an AWS service through an API Gateway API only.

Control Access for Managing an API

In this section, you will learn how to write up IAM policy statements to control who can or cannot create, deploy and update an API in API Gateway. You'll also find the policy statements reference, including the formats of the `Action` and `Resource` fields related to the API managing service.

Control Who Can Create and Manage an API Gateway API with IAM Policies

To control who can or cannot create, deploy and update your API using the API managing service of API Gateway, create an IAM policy document with required permissions as shown in the following policy template:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Permission",  
            "Action": [  
                "apigateway:HTTP_VERB"  
            ],  
            "Resource": [  
                "arn:aws:apigateway:region::resource1-path",  
                "arn:aws:apigateway:region::resource2-path",  
                ...  
            ]  
        }  
    ]  
}
```

Here, *Permission* can be `Allow` or `Deny` to grant or revoke, respectively, the access rights as stipulated by the policy statement. For more information, see [AWS IAM permissions](#).

HTTP_VERB can be any of the [API Gateway-supported HTTP verbs \(p. 255\)](#). * can be used to denote any of the HTTP verbs.

`Resource` contains a list of ARNs of the affected API entities, including [RestApi](#), [Resource](#), [Method](#), [Integration](#), [DocumentationPart](#), [Model](#), [Authorizer](#), [UsagePlan](#), etc. For more information, see [Resource Format of Permissions for Managing API in API Gateway \(p. 256\)](#).

By combining different policy statements, you can customize the access permissions for individual users, groups or roles to access selected API entities and to perform specified actions against those entities. For example, you can include the following statement in the IAM policy to grant your documentation team the permissions to create, publish, update and delete the [documentation parts](#) of a specified API as well as to view the API entities.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "apigateway:GET"  
            ],  
            "Resource": [  
                "arn:aws:apigateway:region::/restapis/api-id/*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "apigateway:PUT",  
                "apigateway:DELETE",  
                "apigateway:POST"  
            ],  
            "Resource": [  
                "arn:aws:apigateway:region::/restapis/api-id/*"  
            ]  
        }  
    ]  
}
```

```
{
    "Effect": "Allow",
    "Action": [
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:DELETE"
    ],
    "Resource": [
        "arn:aws:apigateway:region::/restapis/api-id/documentation/*"
    ]
}
]
```

For your API core development team who is responsible for all operations, you can include the following statement in the IAM policy to grant the team much broader access permissions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:*"
            ],
            "Resource": [
                "arn:aws:apigateway:*::/*"
            ]
        }
    ]
}
```

Statement Reference of IAM Policies for Managing API in API Gateway

The following information describes the **Action** and **Resource** element format used in an IAM policy statement to grant or revoke permissions for managing API Gateway API entities.

Action Format of Permissions for Managing API in API Gateway

The API-managing **Action** expression has the following general format:

```
apigateway:action
```

where **action** is one of the following API Gateway actions:

- *, which represents all of the following actions.
- **GET**, which is used to get information about resources.
- **POST**, which is primarily used to create child resources.
- **PUT**, which is primarily used to update resources (and, although not recommended, can be used to create child resources).
- **DELETE**, which is used to delete resources.
- **PATCH**, which can be used to update resources.
- **HEAD**, which is the same as GET but does not return the resource representation. HEAD is used primarily in testing scenarios.

- **OPTIONS**, which can be used by callers to get information about available communication options for the target service.

Some examples of the Action expression include:

- **apigateway:*** for all API Gateway actions.
- **apigateway:GET** for just the GET action in API Gateway.

Resource Format of Permissions for Managing API in API Gateway

The API-managing Resource expression has the following general format:

```
arn:aws:apigateway:region::resource-path-specifier
```

where *region* is a target AWS region (such as **us-east-1** or ***** for all supported AWS regions), and *resource-path-specifier* is the path to the target resources.

Some example resource expressions include:

- **arn:aws:apigateway:*region*::/restapis/*** for all resources, methods, models, and stages in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/*** for all resources, methods, models, and stages in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/*** for all resources and methods in the resource with the identifier *resource-id*, which is in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/*** for all of the methods in the resource with the identifier *resource-id*, which is in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/resources/*resource-id*/methods/GET** for just the GET method in the resource with the identifier *resource-id*, which is in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/models/*** for all of the models in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/models/*model-name*** for the model with the name of *model-name*, which is in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/stages/*** for all of the stages in the API with the identifier of *api-id* in the AWS region of *region*.
- **arn:aws:apigateway:*region*::/restapis/*api-id*/stages/*stage-name*** for just the stage with the name of *stage-name* in the API with the identifier of *api-id* in the AWS region of *region*.

Control Cross-Account Access to Your API

You can manage access to your APIs by creating IAM permission policies to control who can or cannot create, update, deploy, view, or delete API entities. A policy is attached to an IAM user representing your user, to an IAM group containing the user, or to an IAM role assumed by the user.

In the IAM policies you create for your APIs, you can use Condition elements to allow access only to certain Lambda integrations or authorizers.

The Condition block uses boolean condition operators to match the condition in the policy against values in the request. The String_{Xxx} condition operator will work both for AWS integration (in which the value should be a Lambda function ARN) and Http integration (in which the value should be an Http URI). The following String_{Xxx} condition operators are supported: StringEquals, StringNotEquals, StringEqualsIgnoreCase, StringNotEqualsIgnoreCase, StringLike, StringNotLike. For more information, see [String Condition Operators](#) in the IAM User Guide.

IAM Policy for Cross-Account Lambda Authorizer

Here is an example of an IAM policy to control a cross-account Lambda authorizer function:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST"
            ],
            "Resource": [
                "arn:aws:apigateway:[region]::/restapis/restapi_id/authorizers"
            ],
            //Create Authorizer operation is allowed only with the following Lambda
            function
            "Condition": {
                "StringEquals": {
                    "apigateway:AuthorizerUri":
                    "arn:aws:apigateway:region:lambda:path/2015-03-31/functions/
                    arn:aws:lambda:region:123456789012:function:example/invocations"
                }
            }
        ]
    }
}
```

IAM Policy for Cross-Account Lambda Integration

With cross-account integration, in order to restrict operations on some specific resources (such as put-integration for a specific Lambda function), a Condition element can be added to the policy to specify which resource (Lambda function) is affected.

Here is an example of an IAM policy to control a cross-account Lambda integration function:

To grant another AWS account permission to call [integration:put](#) or [put-integration](#) to set up a Lambda integration in your API, you can include the following statement in the IAM policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:PUT"
            ],
            "Resource": [
                "arn:aws:apigateway:api-region::/restApis/api-id/resources/resource-id/
                methods/GET/integration"
            ],
            //PutIntegration is only valid with the following Lambda function
            "Condition": {
                "StringEquals": {

```

```

        "apigateway:IntegrationUri": "arn:aws:lambda:region:account-
id:function:lambda-function-name"
    }
}
]
}

```

Allow Another Account to Manage the Lambda Function Used When Importing a Swagger File

To grant another AWS account permission to call `restapi:import` or `import-restapi` to import a Swagger file, you can include the following statement in the IAM policy.

In the Condition statement below, the string "lambda:path/2015-03-31/functions/
`arn:aws:lambda:us-east-1:account-id:function:lambda-function-name`" is the full ARN for the Lambda function.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST"
            ],
            "Resource": "arn:aws:apigateway:*/:::restapis",
            "Condition": {
                "StringLike": {
                    "apigateway:IntegrationUri": [
                        "arn:aws:apigateway:apigateway-region:lambda:path/2015-03-31/
functions/arn:aws:lambda:lambda-region:account-id:function:lambda-function-name/
invocations"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST"
            ],
            "Resource": "arn:aws:apigateway:*/:::restapis",
            "Condition": {
                "StringLike": {
                    "apigateway:AuthorizerUri": [
                        "arn:aws:apigateway:apigateway-region:lambda:path/2015-03-31/
functions/arn:aws:lambda:lambda-region:account-id:function:lambda-function-name/
invocations"
                    ]
                }
            }
        }
    ]
}

```

Control Access for Invoking an API

In this section you will learn how to write up IAM policy statements to control who can or cannot 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.

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 \(p. 260\)](#).

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 effectively public accessible.

For example, to grant a user the permission to view a list of pets exposed by a specified API, but to deny the user the 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"  
            ]  
        }  
    ]  
}
```

For a developer team testing APIs, you can include the following statement in the IAM policy to allow the team to call any method on any resource of any API by any developer in the `test` stage.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "execute-api:Invoke",  
                "execute-api:InvalidateCache"  
            ],  
            "Resource": [  
                "arn:aws:execute-api:*:*:*/test/*"  
            ]  
        }  
    ]  
}
```

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. (`*` can be used for all APIs, regardless of the API's identifier.)
- `stage-name` is the name of the stage associated with the method (`*` can be used for all stages, regardless of the stage's name.)
- `HTTP-VERB` is the HTTP verb for the method. It can be one of the following: GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS.
- `resource-path-specifier` is the path to the desired method. (`*` can be used for all paths).

Some example resource expressions include:

- **arn:aws:execute-api:*:::*** for any resource path in any stage, for any API in any AWS region. (This is equivalent to *****).
- **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.
- **arn:aws:execute-api:us-east-1:::*api-id*/test/*/*mydemoresource*/*** for any resource path along the path of **mydemoresource**, for any HTTP method in the stage of test, 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/GET/*mydemoresource*/*** for GET methods under any resource path along the path of **mydemoresource**, in the stage of test, for the API with the identifier of ***api-id*** in the AWS region of us-east-1.

IAM Policy Examples for Managing API Gateway APIs

The following example policy documents shows various use cases to set access permissions for managing API resources in API Gateway. For permissions model and other background information, see [Control Who Can Create and Manage an API Gateway API with IAM Policies \(p. 254\)](#).

Topics

- [Simple Read Permissions \(p. 261\)](#)
- [Read-Only Permissions on any APIs \(p. 262\)](#)
- [Full Access Permissions for any API Gateway Resources \(p. 263\)](#)
- [Full-Access Permissions for Managing API Stages \(p. 263\)](#)
- [Block Specified Users from Deleting any API Resources \(p. 264\)](#)

Simple Read Permissions

The following policy statement gives the user permission to get information about all of the resources, methods, models, and stages in the API with the identifier of **a123456789** in the AWS region of us-east-1:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "apigateway:GET"  
            ],  
            "Resource": [  
                "arn:aws:apigateway:us-east-1::/restapis/a123456789/*"  
            ]  
        }  
    ]  
}
```

The following example policy statement gives the IAM user permission to list information for all resources, methods, models, and stages in any region. The user also has permission to perform all

available API Gateway actions for the API with the identifier of a123456789 in the AWS region of us-east-1:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1:::/restapis/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:)"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1:::/restapis/a123456789/*"
      ]
    }
  ]
}
```

Read-Only Permissions on any APIs

The following policy document will permit attached entities (users, groups or roles) to retrieve any of the APIs of the caller's AWS account. This includes any of the child resources of an API, such as method, integration, etc.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1467321237000",
      "Effect": "Deny",
      "Action": [
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:PATCH",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/*"
      ]
    },
    {
      "Sid": "Stmt1467321341000",
      "Effect": "Deny",
      "Action": [
        "apigateway:GET",
        "apigateway:HEAD",
        "apigateway:OPTIONS"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/",
        "arn:aws:apigateway:us-east-1::/account",
        "arn:aws:apigateway:us-east-1::/clientcertificates",
        "arn:aws:apigateway:us-east-1::/domainnames",
        "arn:aws:apigateway:us-east-1::/apikeys"
      ]
    }
  ]
}
```

```

        ],
    },
    {
        "Sid": "Stmt1467321344000",
        "Effect": "Allow",
        "Action": [
            "apigateway:GET",
            "apigateway:HEAD",
            "apigateway:OPTIONS"
        ],
        "Resource": [
            "arn:aws:apigateway:us-east-1:::restapis/*"
        ]
    }
]
}

```

The first Deny statement explicitly prohibits any calls of POST, PUT, PATCH, DELETE on any resources in API Gateway. This ensures that such permissions will not be overridden by other policy documents also attached to the caller. The second Deny statement blocks the caller to query the root (/) resource, account information (/account), client certificates (/clientcertificates), custom domain names (/domainnames) and API keys (/apikeys). Together, the three statements ensure that the caller can only query API-related resources. This can be useful in API testing when you do not want the tester to modify any of the code.

To restrict the above read-only access to specified APIs, replace the Resource property of Allow statement by the following:

```
"Resource": [ "arn:aws:apigateway:us-east-1:::restapis/restapi_id1/*",
    "arn:aws:apigateway:us-east-1:::restapis/restapi_id2/*"]
```

Full Access Permissions for any API Gateway Resources

The following example policy document grants the full access to any of the API Gateway resource of the AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1467321765000",
            "Effect": "Allow",
            "Action": [
                "apigateway:/*"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

In general, you should refrain from using such a broad and open access policy. It may be necessary to do so for your API development core team so that they can create, deploy, update, and delete any API Gateway resources.

Full-Access Permissions for Managing API Stages

The following example policy documents grants full-access permissions on Stage related resources of any API in the caller's AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:*"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1:::/restapis/*/stages",
                "arn:aws:apigateway:us-east-1:::/restapis/*/*"
            ]
        }
    ]
}
```

The above policy document grants full access permissions only to the `stages` collection and any of the contained stage resources, provided that no other policies granting more accesses have been attached to the caller. Otherwise, you must explicitly deny all the other accesses.

Using the above policy, caller must find out the REST API's identifier beforehand because the user cannot call `GET /restapis` to query the available APIs. Also, if `arn:aws:apigateway:us-east-1:::/restapis/*/*` is not specified in the Resource list, the Stages resource becomes inaccessible. In this case, the caller will not be able to create a stage nor get the existing stages, although he or she can still view, update or delete a stage, provided that he stage's name is known.

To grant permissions for a specific API's stages, simply replace the `restapis/*` portion of the Resource specifications by `restapis/restapi_id`, where `restapi_id` is the identifier of the API of interest.

Block Specified Users from Deleting any API Resources

The following example IAM policy document blocks a specified user from deleting any API resources in API Gateway.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1467331998000",
            "Effect": "Allow",
            "Action": [
                "apigateway:GET",
                "apigateway:HEAD",
                "apigateway:OPTIONS",
                "apigateway:PATCH",
                "apigateway:POST",
                "apigateway:PUT"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1:::/restapis/*"
            ]
        },
        {
            "Sid": "Stmt1467332141000",
            "Effect": "Allow",
            "Action": [
                "apigateway:DELETE"
            ],
            "Condition": {
                "StringNotLike": {

```

```

        "aws:username": "johndoe"
    }
},
"Resource": [
    "arn:aws:apigateway:us-east-1:::/restapis/*"
]
}
}
}
```

This IAM policy grants full access permission to create, deploy, update and delete API for attached users, groups or roles, except for the specified user (johndoe), who cannot delete any API resources. It assumes that no other policy document granting Allow permissions on the root, API keys, client certificates or custom domain names has been attached to the caller.

To block the specified user from deleting specific API Gateway resources, e.g., a specific API or an API's resources, replace the Resource specification above by this:

```
"Resource": ["arn:aws:apigateway:us-east-1:::/restapis/restapi_id_1",  
 "arn:aws:apigateway:us-east-1:::/restapis/restapi_id_2/resources"]
```

IAM Policy Examples for API Execution Permissions

For permissions model and other background information, see [Control Access for Invoking an API \(p. 258\)](#).

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 an IAM User

To enable a user to call the API managing service or the API execution service, you must create an IAM policy for an IAM user, which controls access to the API Gateway entities, and then attach the policy to the IAM user. The following steps describe how to create your IAM policy.

To create your own IAM policy

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. Choose **Policies**, and then choose **Create Policy**. If a **Get Started** button appears, choose it, and then choose **Create Policy**.
3. Next to **Create Your Own Policy**, choose **Select**.
4. For **Policy Name**, type any value that is easy for you to refer to later. Optionally, type descriptive text in **Description**.
5. For **Policy Document**, type a policy statement with the following format, and then choose **Create Policy**:

```
{
  "Version": "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    },
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    }
  ]
}
```

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 IAM user to manage, the API methods the IAM user can call, or both. By default, the IAM user does not have permissions unless there is an explicit corresponding Allow statement.

6. To enable the policy for a user, choose **Users**.
7. Choose the IAM user to whom you want to attach the policy.

You have just created an IAM policy. It won't have any effect until you attach it to an IAM user, to an IAM group containing the user, or to an IAM role assumed by the user.

To attach an IAM policy to an IAM user

1. For the chosen user, choose the **Permissions** tab, and then choose **Attach Policy**.
2. Under **Grant permissions**, choose **Attach existing policies directly**.
3. Choose the policy document just created from the displayed list and then choose **Next: Review**.
4. Under **Permissions summary**, choose **Add permissions**.

Alternatively, you can add the user to an IAM group, if the user is not already a member, and attach the policy document to the group so that the attached policies are applicable to all group members. It is helpful to manage and update policy configurations on a group of IAM users. In the following, we highlight how to attach the policy to an IAM group, assuming that you have already created the group and added the user to the group.

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**.

Enable CORS for an API Gateway Resource

When your API's resources receive requests from a domain other than the API's own domain, you must enable cross-origin resource sharing (CORS) for selected methods on the resource. This amounts to having your API respond to the **OPTIONS** preflight request with at least the following CORS-required response headers:

- **Access-Control-Allow-Methods**
- **Access-Control-Allow-Headers**
- **Access-Control-Allow-Origin**

In API Gateway you enable CORS by setting up an **OPTIONS** method with the mock integration type to return the preceding response headers (with static values discussed in the following) as the method

response headers. In addition, the actual CORS-enabled methods must also return the `Access-Control-Allow-Origin: 'request-originating server addresses'` header in at least its 200 response. You can replace the static value of specific `request-originating server addresses` with `*` to indicate any servers. However, you should be careful of enabling such a broad support and do so only when you fully understand the consequences.

With Lambda, AWS or HTTP integrations, you can leverage API Gateway to set up the required headers using the method response and integration response. For Lambda or HTTP [proxy integrations \(p. 122\)](#), you can still set up the required OPTIONS response headers in API Gateway. However, you must rely on the back end to return the `Access-Control-Allow-Origin` headers because the integration response is disabled for the proxy integration.

Tip

You must set up an OPTIONS method to handle preflight requests to support CORS. However, OPTIONS methods are optional if 1) an API resource exposes only the GET, HEAD or POST methods and 2) the request payload content type is `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain` and 3) the request does not contain any custom headers. When possible, we recommend to use OPTIONS method to enable CORS in your API.

This section describes how to enable CORS for a method in API Gateway using the API Gateway console or the API Gateway [Import an API into API Gateway \(p. 238\)](#).

Topics

- [Prerequisites \(p. 268\)](#)
- [Enable CORS on a Resource Using the API Gateway Console \(p. 268\)](#)
- [Enable CORS on a Resource Using the API Gateway Import API \(p. 270\)](#)

Prerequisites

- You must have the method available in API Gateway. For instructions on how to create and configure a method, see [Build an API with HTTP Custom Integration \(p. 44\)](#).

Enable CORS on a Resource Using the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the API Gateway console, choose an API under **APIs**.
3. Choose a resource under **Resources**. This will enable CORS for all the methods on the resource.
Alternatively, you could choose a method under the resource to enable CORS for just this method.
4. Choose **Enable CORS** from the **Actions** drop-down menu.

The screenshot shows the AWS Lambda interface. On the left, there's a sidebar with 'APIs' and 'HelloWorld' selected. Under 'HelloWorld', 'Resources' is expanded, showing a tree structure with a 'GET /hello' node selected. A context menu is open over this node, with the 'Actions' dropdown expanded. The 'Enable CORS' option is circled in red. To the right of the menu are two panels: 'Method Request' (Auth: NONE, ARN: arn:aws:execute-api:us-east-1:...:vys2ggws7 /GET/hello) and 'Method Response' (HTTP Status: 200, Models: application/json => Empty).

5. In the **Enable CORS** form, do the following:

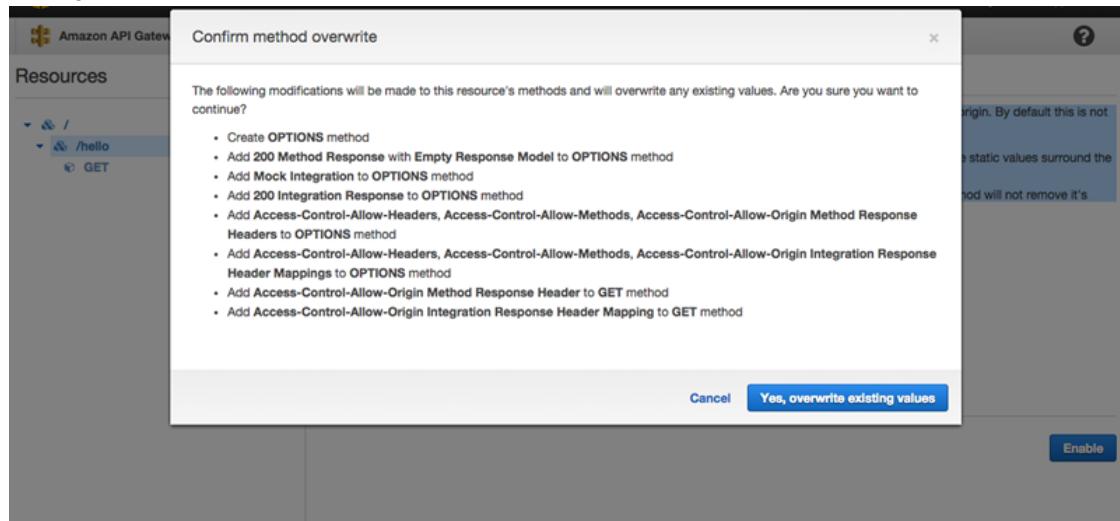
- In the **Access-Control-Allow-Headers** input field, type a static string of a comma-separated list of headers that the client must submit in the actual request of the resource. Use the console-provided header list of 'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token' or specify your own headers.
- Use the console-provided value of '*' as the **Access-Control-Allow-Origin** header value to allow access requests from all domains, or specify a named domain to all access requests from the specified domain.
- Choose **Enable CORS and replace existing CORS headers**.

The screenshot shows the 'Enable CORS' configuration dialog for the '/hello' resource. At the top, the 'Actions' dropdown is shown with 'Enable CORS' highlighted. Below it, a description explains CORS and how to define methods and headers. The 'Methods' section shows 'GET' and 'OPTIONS' checked. The 'Access-Control-Allow-Methods' field contains 'GET,OPTIONS'. The 'Access-Control-Allow-Headers' field contains "'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'". The 'Access-Control-Allow-Origin' field is empty and has a warning icon. At the bottom, a large blue button says 'Enable CORS and replace existing CORS headers'.

Note

When applying the above instructions to the ANY method in a proxy integration, any applicable CORS headers will not be set. Instead, you rely on the integration backend to return the applicable CORS headers, such as Access-Control-Allow-Origin.

6. In **Confirm method changes**, choose **Yes, overwrite existing values** to confirm the new CORS settings.



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 origin domain names, if you do not want to return the Cross-origin access error.

As with any updates of your API, you must deploy or redeploy the API for the new settings to take effect.

Enable CORS on a Resource Using the API Gateway Import API

If you are using the [API Gateway Import API \(p. 238\)](#), you can set up CORS support using a Swagger 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.

The following example creates an OPTIONS method and specifies mock integration. For more information, see [Set up Mock Integrations in API Gateway \(p. 152\)](#).

```
/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:
    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 : "'*'"
      responseTemplates:
        application/json: |
          {}
  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.

```

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:

```

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 : "'*'"

```

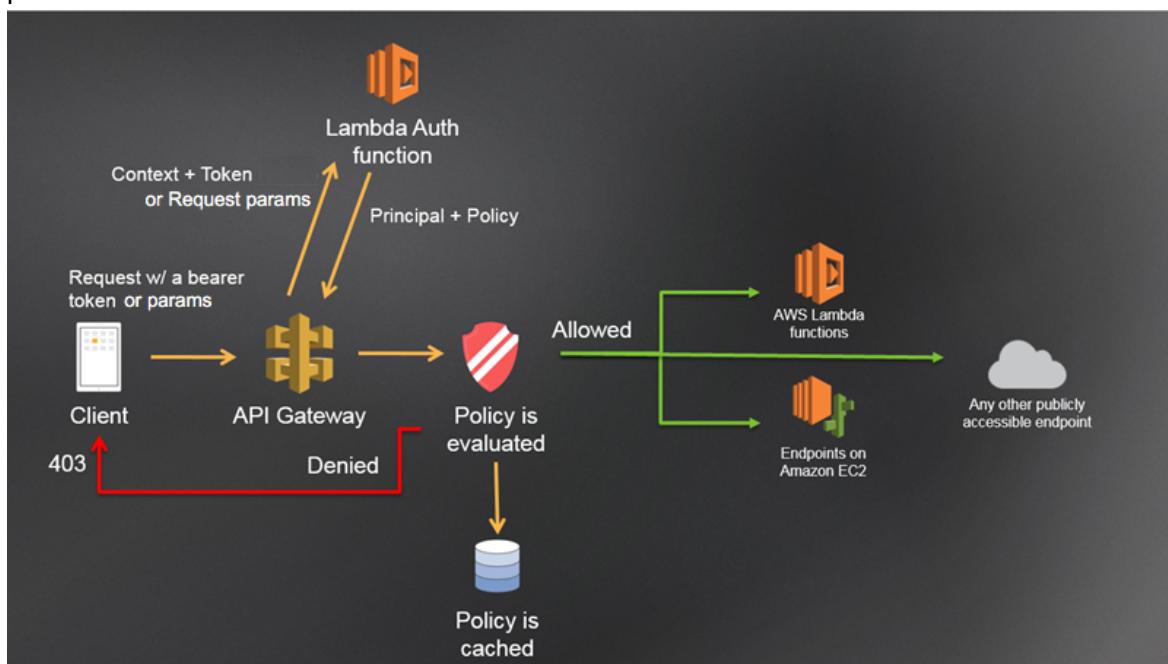
Use API Gateway Lambda Authorizers

An Amazon API Gateway *Lambda authorizer* (formerly known as a *custom authorizer*) is a Lambda function that you provide to control access to your API methods. A Lambda authorizer uses bearer token authentication strategies, such as OAuth or SAML. It can also use information described by headers, paths, query strings, stage variables, or context variables request parameters.

Note

Path parameters can be used to grant or deny permissions to invoke a method, but they cannot be used to define identity sources, which can be used as parts of an authorization policy caching key. Only headers, query strings, stage variables, and context variables can be set as identity sources.

When a client calls your API, API Gateway verifies whether a Lambda authorizer is configured for the API method. If so, API Gateway calls the Lambda function. In this call, API Gateway supplies the authorization token that is extracted from a specified request header for the token-based authorizer, or passes in the incoming request parameters as the input (for example, the event parameter) to the request parameters-based authorizer function.



You can implement various authorization strategies, such as JSON Web Token (JWT) verification and OAuth provider callout. You can also implement a custom scheme based on incoming request parameter values, to return IAM policies that authorize the request. If the returned policy is invalid or the permissions are denied, the API call does not succeed. For a valid policy, API Gateway caches the returned policy, associated with the incoming token or identity source request parameters. It then uses the cached policy for the current and subsequent requests, over a pre-configured time-to-live (TTL) period of up to 3600 seconds. You can set the TTL period to zero seconds to disable the policy caching. The default TTL value is 300 seconds. Currently, the maximum TTL value of 3600 seconds cannot be increased.

Topics

- [Types of API Gateway Lambda Authorizers \(p. 273\)](#)
- [Create an API Gateway Lambda Authorizer Lambda Function \(p. 273\)](#)
- [Input to an Amazon API Gateway Lambda Authorizer \(p. 277\)](#)
- [Output from an Amazon API Gateway Lambda Authorizer \(p. 279\)](#)

- [Configure Lambda Authorizer Using the API Gateway Console \(p. 280\)](#)
- [Call an API with API Gateway Lambda Authorizers \(p. 282\)](#)
- [Configure Cross-Account Lambda Authorizer Using the API Gateway Console \(p. 285\)](#)

Types of API Gateway Lambda Authorizers

API Gateway supports Lambda authorizers of the `TOKEN` and `REQUEST` types:

- Lambda authorizers of the `TOKEN` type grant a caller permissions to invoke a given request using an authorization token passed in a header. The token could be, for example, an OAuth token.
- Lambda authorizers of the `REQUEST` type grant a caller permissions to invoke a given request using request parameters, including headers, query strings, stage variables, or context parameters.

Create an API Gateway Lambda Authorizer Lambda Function

Before creating an API Gateway Lambda authorizer, you must first create the AWS Lambda function that implements the logic to authorize and, if necessary, to authenticate the caller. You can do so in the Lambda console, using the code template available from the API Gateway Lambda Authorizer blueprint. Or you can create one from scratch, following this [example in awslabs](#). For illustration purposes, we explain how to create a simple Lambda function from scratch without using a blueprint. In production code, you should follow the API Gateway Lambda Authorizer blueprint to implement your authorizer Lambda function.

When creating the Lambda function for your API Gateway Lambda authorizer, you assign an execution role for the Lambda function if it calls other AWS services. For the following example, the basic `AWSLambdaRole` suffices. For more involved use cases, follow the [instructions](#) to grant permissions in an execution role for the Lambda function.

Control a Lambda Function of a Lambda Authorizer

To grant another AWS account permission to call `authorizer:create` or `create-authorizer` to control the Lambda function used in your Lambda authorizer, you can create the following IAM policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "apigateway:POST"  
            ],  
            "Resource": [  
                "arn:aws:apigateway:region::/restapis/restapi_id/authorizers"  
            ],  
            //Create Authorizer operation is allowed only with the following Lambda  
            "function":  
                "Condition": {  
                    "StringEquals": {  
                        "apigateway:AuthorizerUri": "arn:aws:lambda:region:account-  
                        id:function:"  
                    }  
                }  
        }  
    ]  
}
```

```
}
```

Create a Lambda Function for a Lambda Authorizer of the TOKEN type

In the code editor of the Lambda console, enter the following Node.js code as an example of the API Gateway Lambda authorizer of the TOKEN type.

```
// A simple TOKEN 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', the
// function
// returns the 'Unauthorized' error with an HTTP status code of 401. For any other token
// value,
// the authorizer returns an 'Invalid token' error.

exports.handler = function(event, context, callback) {
    var token = event.authorizationToken;
    switch (token.toLowerCase()) {
        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");
    }
};

// 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;
}
```

For Lambda authorizers of the `TOKEN` type, API Gateway passes the source token to the Lambda function as the `event.authorizationToken`. Based on the value of this token, the preceding authorizer function returns an Allow IAM policy on a specified method if the token value is `'allow'`. This permits a caller to invoke the specified method. The caller receives a `200 OK` response. The authorizer function returns a Deny policy against the specified method if the authorization token has a `'deny'` value. This blocks the caller from calling the method. The client receives a `403 Forbidden` response. If the token is `'unauthorized'`, the client receives a `401 Unauthorized` response. If the token is `'fail'` or anything else, the client receives a `500 Internal Server Error` response. In both of the last two cases, no IAM policy is generated and the calls fail.

Note

In production code, you may need to authenticate the user before granting authorizations. If so, you can add authentication logic in the Lambda function as well. Consult the provider-specific documentation for instructions on how to call such an authentication provider.

In addition to returning an IAM policy, the Lambda authorizer function must also return the caller's principal identifier. It can optionally return a key-value map named `context`, containing additional information that can be passed into the integration backend. For more information about the authorizer's output format, see [Output from an Amazon API Gateway Lambda Authorizer \(p. 279\)](#).

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`. For the preceding Lambda authorizer example, `key` is `stringKey`, `numberKey`, or `booleanKey`. Their values are stringified, for example, `"stringval"`, `"123"`, or `"true"`, respectively.

Before going further, you may want to test the Lambda function from within the Lambda console. To do this, configure the sample event to provide an input as described in [Input to an Amazon API Gateway Lambda Authorizer \(p. 277\)](#) and verify the result by examining the output compatible with [Output from an Amazon API Gateway Lambda Authorizer \(p. 279\)](#). The next subsection explains how to create a Lambda function of the Request authorizer.

Create a Lambda Function of a Lambda Authorizer of the REQUEST type

In the code editor of the Lambda console, enter the following Node.js code for a simplified Lambda function as an example of the API Gateway Lambda authorizers of the `REQUEST` type.

```
exports.handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));

    // 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, QueryString1 query parameter,
    // stage variable of StageVar1 and the accountId in the request context all match
    // specified values of 'headerValue1', 'queryValue1', 'stageValue1', and
    // '123456789012', respectively.

    // Retrieve request parameters from the Lambda function input:
    var headers = event.headers;
    var queryStringParameters = event.queryStringParameters;
    var pathParameters = event.pathParameters;
    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 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 = {};
conditionIpAddress = {};

if (headers.HeaderAuth1 === "headerValue1"
    && queryStringParameters.QueryString1 === "queryValue1"
    && stageVariables.StageVar1 === "stageValue1"
    && requestContext.accountId === "123456789012") {
    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);
}

```

This Lambda function of the REQUEST authorizer verifies the input request parameters to return an Allow IAM policy on a specified method if all the required parameter (HeaderAuth1, QueryString1,

`StageVar1`, and `accountId`) values match the pre-configured ones. This permits a caller to invoke the specified method. The caller receives a 200 OK response. Otherwise, the authorizer function returns an Unauthorized error, without generating any IAM policy.

The above example authorizer function in Node.js illustrates the programming flow to create a Lambda authorizer of the REQUEST type, including parsing the input which is similar to parsing the [Lambda function input in the Lambda proxy integration \(p. 132\)](#). You can extend the implementation to other languages supported by Lambda, such as Java or Python. For example to parse the input to a Lambda REQUEST authorizer in Java, see [the section called “Java Function for an API with Lambda Proxy Integration” \(p. 21\)](#).

Before going further, you may want to test the Lambda function from within the Lambda console. To do this, configure the sample event to provide the input and verify the result by examining the output. The next two sections explain the [Input to an Amazon API Gateway Lambda Authorizer \(p. 277\)](#) and [Output from an Amazon API Gateway Lambda Authorizer \(p. 279\)](#).

Input to an Amazon API Gateway Lambda Authorizer

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 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>/<method>/<resourcePath>"  
}
```

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. The `methodArn` is the ARN of the incoming method request and is populated by API Gateway in accordance with the Lambda authorizer configuration.

For the example TOKEN authorizer Lambda function shown in the preceding section, the `<caller-supplied-token>` string is `allow`, `deny`, `unauthorized`, or any other string value. An empty string value is the same as `unauthorized`. The following shows an example of such an input to obtain an Allow policy on the GET method of an API (`ymy8tbxw7b`) of the AWS account (123456789012) in any stage (*).

```
{  
  "type": "TOKEN",  
  "authorizationToken": "allow",  
  "methodArn": "arn:aws:execute-api:us-west-2:123456789012:ymy8tbxw7b/*/GET/"  
}
```

For a Lambda authorizer of the REQUEST type, API Gateway passes the required request parameters to the authorizer Lambda function as part of the event object. The affected 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.

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:s4x3opwd6i/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": "...",
        "X-Forwarded-Proto": "https",
        "CloudFront-Is-SmartTV-Viewer": "false",
        "Host": "...execute-api.us-east-1.amazonaws.com",
        "Accept-Encoding": "gzip, deflate",
        "X-Forwarded-Port": "443",
        "X-Amzn-Trace-Id": "...",
        "Via": "...cloudfront.net (CloudFront)",
        "X-Amz-Cf-Id": "...",
        "X-Forwarded-For": "..., ...",
        "Postman-Token": "...",
        "cache-control": "no-cache",
        "CloudFront-Is-Desktop-Viewer": "true",
        "Content-Type": "application/x-www-form-urlencoded"
    },
    "queryStringParameters": {
        "QueryString1": "queryValue1"
    },
    "pathParameters": {},
    "stageVariables": {
        "StageVar1": "stageValue1"
    },
    "requestContext": {
        "path": "/request",
        "accountId": "123456789012",
        "resourceId": "05c7jb",
        "stage": "test",
        "requestId": "...",
        "identity": {
            "apiKey": "...",
            "sourceIp": ...
        },
        "resourcePath": "/request",
        "httpMethod": "GET",
        "apiId": "s4x3opwd6i"
    }
}
```

The `requestContext` is a map of key-value pairs and corresponds to the [\\$context \(p. 191\)](#) variable. Its outcome is API-dependent. API Gateway may add new keys to the map. For more information about the Lambda function input in a proxy integration, see [Input Format of a Lambda Function for Proxy Integration \(p. 132\)](#).

Output from an Amazon API Gateway Lambda Authorizer

The Lambda authorizer's Lambda function returns an output that 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 has a usage plan enacted, the method requires an API key, and the `apiKeySource` is set to `AUTHORIZER`, the Lambda authorizer Lambda function must also return an API key of the usage plan 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}:{appId}/{stage}/{httpVerb}/[{resource}]/[child-resources]"  
            }  
        ]  
    },  
    "context": {  
        "stringKey": "value",  
        "numberKey": "1",  
        "booleanKey": "true"  
    },  
    "usageIdentifierKey": "{api-key}"  
}
```

Here, a policy statement stipulates 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 \(p. 260\)](#).

For an authorization-enabled method AR, e.g., `arn:aws:execute-api:{region-id}:{account-id}:{api-id}/{stage-id}/{method}/{resource}/{path}`, 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 `$context.authorizer.principalId` variable. This is useful if you want to pass the value to the backend. For more information, see [Accessing the \\$context Variable \(p. 191\)](#).

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.

{api-key} stands for an API key of the usage plan associated with the API stage. For more information, see [the section called “Use API Gateway Usage Plans” \(p. 314\)](#).

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 in an API (y my8tbxw7b) of an AWS account (123456789012) in any stage (*).

```
{  
    "principalId": "user",  
    "policyDocument": {  
        "Version": "2012-10-17",  
        "Statement": [  
            {  
                "Action": "execute-api:Invoke",  
                "Effect": "Deny",  
                "Resource": "arn:aws:execute-api:us-west-2:123456789012:y my8tbxw7b/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*
```

Configure 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 enable a Lambda authorizer on API methods

1. Sign in to the API Gateway console.
2. Create a new or select an existing API and choose **Authorizers** under that API.
3. Choose **Create New Authorizer**.
4. For **Create Authorizer**, type an authorizer name in the **Name** input field.
5. For **Type**, choose the **Lambda** option.
6. For **Lambda Function**, choose a region and then choose an available Lambda authorizer function that's in your account.
7. Leave **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 type the name of an IAM role to allow API Gateway to invoke the authorizer Lambda function. For an example of such a role, see [Set Up an IAM Role and Policy for an API to Invoke Lambda Functions \(p. 509\)](#).

If you choose to let the API Gateway console set the resource-based policy, the **Add Permission to Lambda Function** dialog is displayed. Choose **OK**. After the Lambda authorization is created, you can test it with appropriate authorization token values to verify that it works as expected.

8. For **Lambda Event Payload**, choose 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** options, do the following:
 - Type the name of a header in **Token Source**. The API client must include a header of this name to send the authorization token to the Lambda authorizer.
 - Optionally, provide a RegEx statement in **Token Validation** input field. API Gateway performs initial validation of the input token against this expression and invokes the

authorizer upon successful validation. This helps reduce chances of being charged for invalid tokens.

- For **Authorization Caching**, select or clear the **Enabled** option, depending on whether you want to cache the authorization policy generated by the authorizer or not. When policy caching is enabled, you can choose to modify the **TTL** value from the default (300). Setting **TTL=0** disables policy caching. When policy caching is enabled, the header name specified in **Token Source** becomes the cache key.
- b. For the **Request** option, do the following:

- For **Identity Sources**, type a request parameter name of a chosen parameter type. Supported parameter types are **Header**, **Query String**, **Stage Variable**, and **Context**. To add more identity sources, choose **Add Identity Source**.

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 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.

- For **Authorization Caching**, select or deselect the **Enabled** option, depending on whether you want to cache the authorization policy generated by the authorizer or not. When policy caching is enabled, you can choose to modify the **TTL** value from the default (300). Setting **TTL=0** disables policy caching.

When caching is disabled, it is not necessary to specify an identity source. API Gateway does not perform any validation before invoking the authorizer's 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 set the TTL value to zero to disable policy caching for the API.

10. Choose **Create** to create the new Lambda authorizer for the chosen API.
11. After the authorizer is created for the API, you can optionally test invoking the authorizer before it is configured on a method.

For the **TOKEN** authorizer, type a valid token in the **Identity token** input text field and the choose **Test**. The token will be passed to the Lambda function as the header you specified in the **Identity token source** setting of the authorizer.

For the **REQUEST** authorizer, type the valid request parameters corresponding to the specified identity sources and then choose **Test**.

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 Method \(p. 458\)](#). 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. Go back to the API. Create a new method or choose an existing method. If necessary, create a new resource.
2. In **Method Execution**, choose the **Method Request** link.
3. Under **Settings**, expand the **Authorization** drop-down list to select the Lambda authorizer you just created (for example, `myTestApiAuthorizer`), and then choose the check mark icon to save the choice.
4. Optionally, while still on the **Method Request** page, choose **Add header** if you also want to pass the authorization token to the backend. In **Name**, type a header name that matches the **Token Source** name you specified when you created the Lambda authorizer for the API. Then, choose the check mark icon to save the settings. This step does not apply to `REQUEST` authorizers.
5. 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 in **Stage Editor**.

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, null, or 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, null or empty.

In the following, we show how to use Postman to call or test the API with the previously described Lambda `TOKEN` authorizer enabled. 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**.

The screenshot shows the Postman interface with a GET request to `https://<api-id>.execute-api.<region>.amazonaws.com/test`. The 'Auth' tab under 'Authorization' is selected with the value 'allow'. The response status is **200 OK**, and the response body is a JSON object containing various headers and parameters, including 'args': {}, 'headers': {}, and 'url': 'http://httpbin.org/get'.

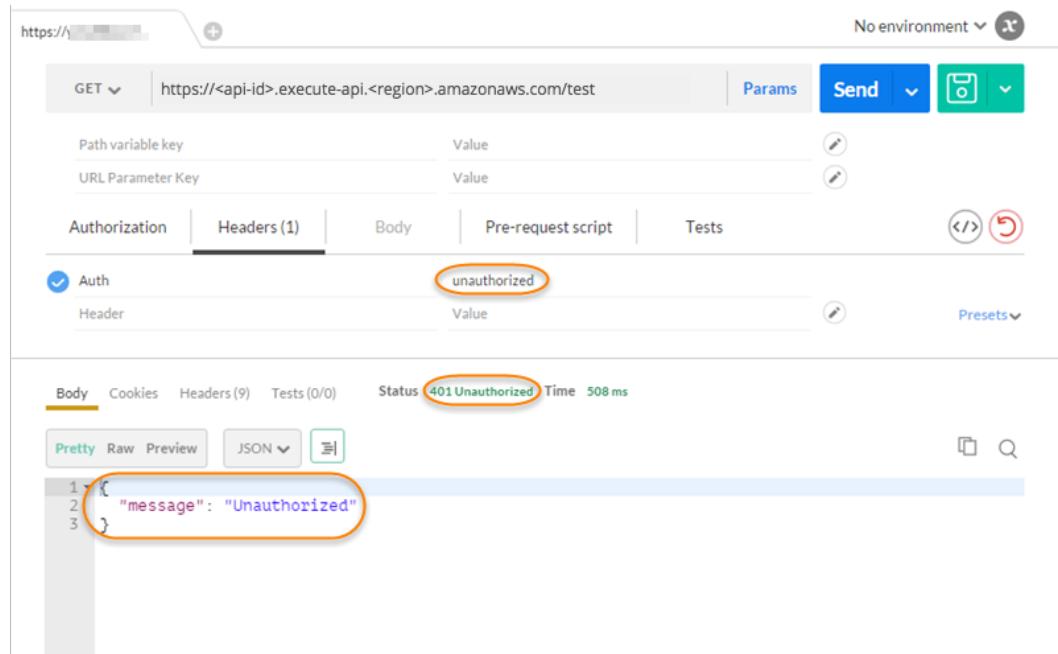
The response shows that the API Gateway Lambda authorizer returns a **200 OK** response and successfully authorizes the call to access the HTTP endpoint (`http://httpbin.org/get`) integrated with the method.

2. Still in Postman, change the Lambda authorization token header value to **deny**. Choose **Send**.

The screenshot shows the Postman interface with the same GET request. The 'Auth' tab under 'Authorization' is selected with the value 'denied'. The response status is **403 Forbidden**, and the response body is a JSON object with a single key 'Message' containing the value 'User is not authorized to access this resource'.

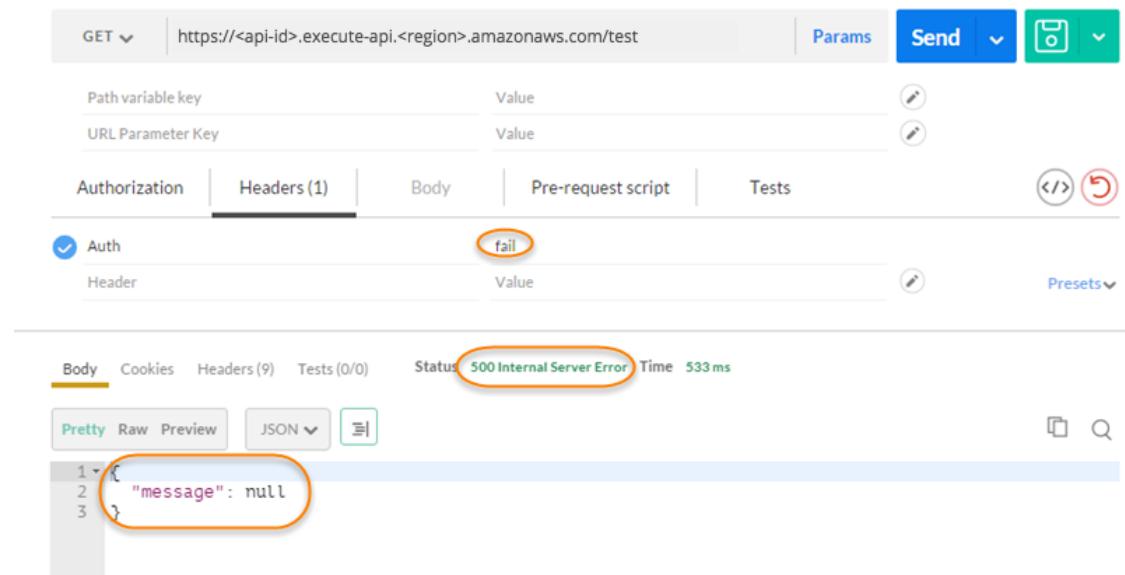
The response shows that the API Gateway Lambda authorizer returns a **403 Forbidden** response without authorizing the call to access the HTTP endpoint.

3. In Postman, change the Lambda authorization token header value to **unauthorized** and choose **Send**.



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**.



The response shows that API Gateway returns a **500 Internal Server Error** response without authorizing the call to access the HTTP endpoint.

Configure Cross-Account Lambda Authorizer Using the API Gateway Console

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 functions 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 Cross-Account Lambda Authorizer Using the API Gateway Console

Log in to the Amazon API Gateway console in your first account (the one that has your API in it) and do the following:

1. Locate your API and choose **Authorizers**.
2. Choose **Create New Authorizer**.
3. For **Create Authorizer**, type an authorizer name in the **Name** input field.
4. For **Type**, choose the **Lambda** option.
5. For **Lambda Function**, copy-paste 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. Leave **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 type the name of an IAM role to allow API Gateway to invoke the authorizer Lambda function. For an example of such a role, see [Set Up an IAM Role and Policy for an API to Invoke Lambda Functions \(p. 509\)](#).

If you choose to let the API Gateway console set the resource-based policy, the **Add Permission to Lambda Function** dialog is displayed. Choose **OK**. After the Lambda authorization is created, you can test it with appropriate authorization token values to verify that it works as expected.

7. For **Lambda Event Payload**, choose either **Token** for a TOKEN authorizer or **Request** for a REQUEST authorizer.
8. Depending on the choice you made in the previous step, do one of the following:
 - a. For the **Token** options, do the following:
 - i. Type the name of a header in **Token Source**. The API client must include a header of this name to send the authorization token to the Lambda authorizer.
 - ii. Optionally, provide a RegEx statement in **Token Validation** input field. API Gateway performs initial validation of the input token against this expression and invokes the authorizer upon successful validation. This helps reduce chances of being charged for invalid tokens.
 - iii. For **Authorization Caching**, select or clear the **Enabled** option, depending on whether you want to cache the authorization policy generated by the authorizer or not. When policy

caching is enabled, you can choose to modify the **TTL** value from the default (300). Setting TTL=0 disables policy caching. When policy caching is enabled, the header name specified in **Token Source** becomes the cache key.

- b. For the **Request** option, do the following:

- i. For **Identity Sources**, type a request parameter name of a chosen parameter type. Supported parameter types are **Header**, **Query String**, **Stage Variable**, and **Context**. To add more identity sources, choose **Add Identity Source**.

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.

- ii. For **Authorization Caching**, leave the **Enabled** option selected. Leave the **TTL** value set to the default (300).

9. Choose **Create** to create the new Lambda authorizer for the chosen API.
10. You'll see a popup that says **Add Permission to Lambda Function: You have selected a Lambda function from another account. Please ensure that you have the appropriate Function Policy on this function. You can do this by running the following AWS CLI command from account 123456789012**, followed by an aws lambda add-permission command string.
11. Copy-paste the aws lambda add-permission command string into an AWS CLI window that is configured for your second account. This will grant your first account access to your second account's Lambda authorizer function.
12. In the popup from the previous step, choose **OK**.

Use Amazon Cognito User Pools

In addition to using [IAM roles and policies \(p. 252\)](#) or [Lambda authorizers \(p. 272\)](#) (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.
- 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](#). Set the authorization type of your API 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 User Pool Authorizers \(p. 287\)](#)
- [Create an Amazon Cognito User Pool \(p. 288\)](#)
- [Integrate an API with a User Pool \(p. 289\)](#)
- [Call an API Integrated with a User Pool \(p. 293\)](#)

Obtain Permissions to Create User Pool Authorizers

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:ResourceArn": [  
                        "arn:aws:cognito-idp:us-east-1:1234567890123:userpool/us-east-1_xJ1MQtPEN"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        "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 your IAM user, 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

Before integrating your API with a user pool, you must create the user pool in Amazon Cognito. For instructions on how to create a user pool, see [Setting up User Pools](#) in the *Amazon Cognito Developer Guide*.

Note

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, 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

General settings

Users and groups

Attributes

Policies

MFA and verifications

Advanced security beta

Message customizations

Tags

Devices

App clients

Triggers

Analytics

App integration

App client settings

Domain name

UI customization

Resource servers

Federation

Identity providers

Attribute mapping

Do you want to define resource servers and custom scopes?

You can define resource servers and custom scopes for OAuth 2.0 flows. [Learn more about resource servers and scopes.](#)

hamuta movies

Identifier com.haymuta.movies

Scopes

Name	Description
drama.view	sdfa
comedy.view	asdfad
e.g., weather.read	e.g., Read weather forecasts

Add another resource server

Configure app client settings

Integrate an API with a 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 walks you through the steps to do this using the API Gateway console.

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. From the main navigation pane, choose **Authorizers** under the specified API.
3. Under **Authorizers**, choose **Create New Authorizer**.
4. To configure the new authorizer to use a user pool, do the following:
 - a. Type an authorizer name in **Name**.
 - b. Select the **Cognito** option.
 - c. Choose a region under **Cognito User Pool**.
 - d. Select an available user pool. You must have created a user pool for the selected region in Amazon Cognito for it to show up in the drop-down list.
 - e. For **Token source**, type **Authorization** as the header name to pass the identity or access token that's returned by Amazon Cognito when a user signs in successfully.
 - f. Optionally, type a regular expression in the **Token validation** field to validate the aud field of the identity token before the request is authorized with Amazon Cognito.
 - g. To finish integrating the user pool with the API, choose **Create**.
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. Make sure to use the returned identity token, not the access 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. The next procedure walks you through the steps to configure the authorizer on an API method.

To configure a COGNITO_USER_POOLS authorizer on methods

1. Choose (or create) a method of your API.
2. Choose **Method Request**.
3. Under **Settings**, choose the pencil icon next to **Authorization**.
4. Choose one of the available **Amazon Cognito user pool authorizers** from the drop-down list.
5. To save the settings, choose the check mark icon.
6. To use an identity token, do the following:
 - a. Leave the **OAuth Scopes** option unspecified (as **NONE**).
 - b. If needed, choose **Integration Request** to 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']"
  }
}
```

7. To use an access token, do the following:
 - a. Choose the pencil icon next to **OAuth Scopes**.
 - b. Type 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 \(p. 288\)](#), one of the scopes is `com.hamuta.movies/drama.view`. Use a single space to separate multiple scopes.

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.

- c. To save the setting, choose the check mark icon.
8. 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 a Swagger definition file and importing the API definition into API Gateway.

To import a COGNITO_USER_POOLS authorizer with a Swagger definition file

1. Create (or export) a Swagger definition file for your API.
2. Specify the COGNITO_USER_POOLS authorizer (`MyUserPool`) definition as part of the `securityDefinitions`:

```
"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/scopes, ...] }:

```

"paths": {
    "/": {
        "get": {
            "consumes": [
                "application/json"
            ],
            "produces": [
                "text/html"
            ],
            "responses": {
                "200": {
                    "description": "200 response",
                    "headers": {
                        "Content-Type": {
                            "type": "string"
                        }
                    }
                }
            },
            "security": [
                {
                    "MyUserPool": ["com.hamuta.movies/drama.view", "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 Swagger definitions or extensions. For more information, see [API Gateway Extensions to Swagger \(p. 486\)](#).

Call an API Integrated with a 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 token of the signed-in user from the user pool.
- Include the identity token in the `Authorization` header (or another header you specified when you created the authorizer).

You can use one of the [AWS SDKs](#) to perform these tasks. For example:

- To use the Android SDK, see [Setting up the AWS Mobile SDK for Android to Work with User Pools](#).
- To use the iOS SDK, see [Setting Up the AWS Mobile SDK for iOS to Work with User Pools](#).
- To use JavaScript, see [Setting up the AWS SDK for JavaScript in the Browser to Work with User Pools](#).

The following procedure outlines the steps to perform these tasks. For more information, see the blog posts on [Using Android SDK with Amazon Cognito User Pools](#) and [Using Amazon Cognito User Pool for iOS](#).

To call an API that's integrated with a user pool

1. Sign up a first-time user to a specified user pool.
2. Sign in a user to the user pool.
3. Get the user's identity token.
4. Call API methods that are configured with a user pool authorizer, and supply the unexpired token in the `Authorization` header or another header of your choosing.
5. When the token expires, repeat steps 2–4. Identity tokens provisioned by Amazon Cognito expire within an hour.

For code examples, see an [Android Java sample](#) and an [iOS Objective-C sample](#).

Use Client-Side SSL Certificates for Authentication by the Backend

You can use API Gateway to generate an SSL certificate and 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 originating from Amazon API Gateway, even if the backend is publicly accessible.

Note

Some backend servers may not support SSL client authentication as API Gateway does and could return an SSL certificate error. For a list of incompatible backend servers, see [Known Issues \(p. 585\)](#).

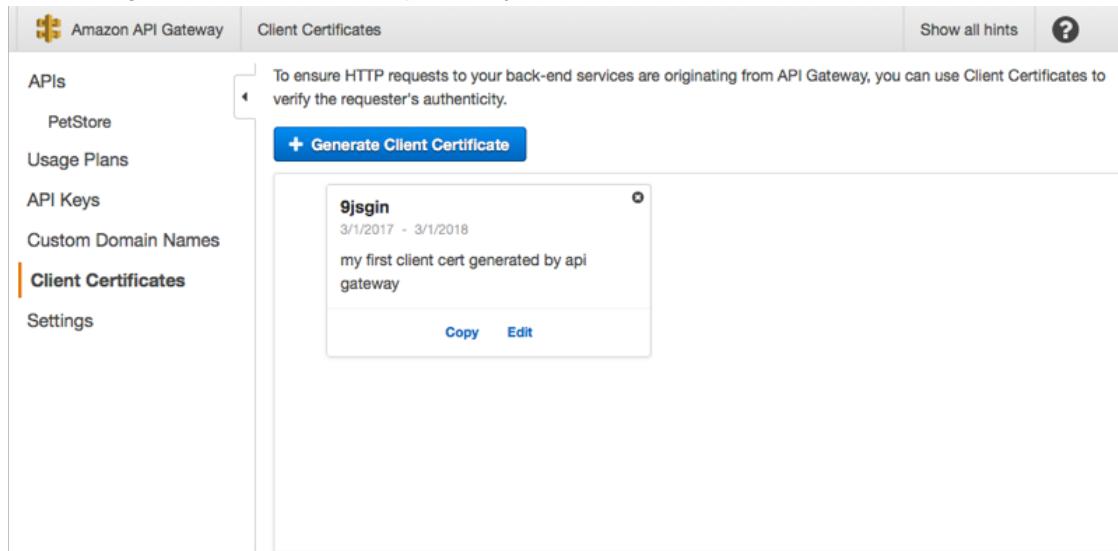
The SSL certificates that are generated by API Gateway are self-signed and only the public key of a certificate is visible in the API Gateway console or through the APIs.

Topics

- [Generate a Client Certificate Using the API Gateway Console \(p. 294\)](#)
- [Configure an API to Use SSL Certificates \(p. 294\)](#)
- [Test Invoke to Verify the Client Certificate Configuration \(p. 295\)](#)
- [Configure Backend HTTPS Server to Verify the Client Certificate \(p. 297\)](#)
- [Rotate an Expiring Client Certificate \(p. 297\)](#)
- [API Gateway-Supported Certificate Authorities for HTTP and HTTP Proxy Integrations \(p. 298\)](#)

Generate a Client Certificate Using the API Gateway Console

1. In the main navigation pane, choose **Client Certificates**.
2. From the **Client Certificates** pane, choose **Generate Client Certificate**.
3. Optionally, for **Edit**, choose to add a descriptive title for the generated certificate and choose **Save** to save the description. API Gateway generates a new certificate and returns the new certificate GUID, along with the PEM-encoded public key.



You are now ready to configure an API to use the certificate.

Configure an API to Use SSL Certificates

These instructions assume you already completed [Generate a Client Certificate Using the API Gateway Console \(p. 294\)](#).

1. In the API Gateway console, create or open an API for which you want to use the client certificate. Make sure the API has been deployed to a stage.
2. Choose **Stages** under the selected API and then choose a stage.
3. In the **Stage Editor** panel, select a certificate under the **Client Certificate** section.
4. To save the settings, choose **Save Changes**.

prod Stage Editor Delete Stage

Invoke URL: <https://bh1tfjis4h.execute-api.us-east-1.amazonaws.com/prod>

Settings **Stage Variables** **SDK Generation** **Export** **Deployment History** **Documentation History**

Configure the metering and caching settings for the **prod** stage.

Cache Settings

Enable API cache

CloudWatch Settings

Enable CloudWatch Logs ⓘ

Enable Detailed CloudWatch Metrics ⓘ

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is **500** requests per second with a burst of **1000** requests. ⓘ

Enable throttling ⓘ

Rate requests per second

Burst requests

Client Certificate

Select the client certificate that API Gateway will use to call your integration endpoints in this stage.

Certificate

Save Changes

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. In **Client**, choose **Test**.
2. From **Client Certificate**, choose **Test** to invoke the method request.

[← Method Execution](#)

/ - GET - Method Test

Make a test call to your method with the provided input

Path

No path parameters exist for this resource. You can define path parameters by using the syntax **{myPathParam}** in a resource path.

Query Strings

No query string parameters exist for this method. You can add them via Method Request.

Headers

No header parameters exist for this method. You can add them via Method Request.

Stage Variables

No [stage variables](#) exist for this method.

Client Certificate

my first client cert generated by api gate ▾

Request Body

Request Body is not supported for GET methods.

API Gateway presents the chosen SSL certificate for the HTTP backend to authenticate the API.

Configure Backend HTTPS Server to Verify the Client Certificate

These instructions assume you already completed [Generate a Client Certificate Using the API Gateway Console \(p. 294\)](#) 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" (p. 298).

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-crt.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-crt.pem'),
  ca: fs.readFileSync('apig-crt.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, follow the steps below:

1. Generate a new client certificate, by calling `clientcertificate:create` of the API Gateway REST API or the AWS CLI command of `create-client-certificate`. For purposes of discussions, we assume the new client certificate ID is `ndiqef`.
2. Update the backend server to include the new client certificate. Do not remove the existing client certificate yet.

Some server may 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 (ndiqgef):

```
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" : "ndiqgef"
    }
  ]
}
```

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 clientCertificateld (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
```

API Gateway-Supported Certificate Authorities for HTTP and HTTP Proxy Integrations

The following list shows the certificate authorities supported by API Gateway for HTTP and HTTP Proxy integrations.

```
Alias name: mozillacert81.pem
MD5: D5:E9:81:40:C5:18:69:FC:46:2C:89:75:62:0F:AA:78
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:C4:40:8E

Alias name: mozillacert99.pem
MD5: 2B:70:20:56:86:82:A0:18:C8:07:53:12:28:70:21:72
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:50

Alias name: swisssignplatinumg2ca
MD5: C9:98:27:77:28:1E:3D:0E:15:3C:84:00:B8:85:03:E6
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:14:36

Alias name: mozillacert145.pem
MD5: 60:84:7C:5A:CE:DB:0C:D4:CB:A7:E9:FE:02:C6:A9:C0
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:A3
```

```
Alias name: mozillacert37.pem
MD5: AB:57:A6:5B:7D:42:82:19:B5:D8:58:26:28:5E:FD:FF
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:A1:2D
Alias name: mozillacert4.pem
MD5: 4F:EB:F1:F0:70:C2:80:63:5D:58:9F:DA:12:3C:A9:C4
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:3F
Alias name: mozillacert70.pem
MD5: 5E:80:9E:84:5A:0E:65:0B:17:02:F3:55:18:2A:3E:D7
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:C0
Alias name: mozillacert88.pem
MD5: 73:9F:4C:4B:73:5B:79:E9:FA:BA:1C:EF:6E:CB:D5:C9
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:71
Alias name: mozillacert134.pem
MD5: FC:11:B8:D8:08:93:30:00:6D:23:F9:7E:EB:52:1E:02
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:D1:22
Alias name: mozillacert26.pem
MD5: DC:32:C3:A7:6D:25:57:C7:68:09:9D:EA:2D:A9:A2:D1
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:73
Alias name: buypassclass2ca
MD5: 46:A7:D2:FE:45:FB:64:5A:A8:59:90:9B:78:44:9B:29
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:48
Alias name: chunghwaepkirootca
MD5: 1B:2E:00:CA:26:06:90:3D:AD:FE:6F:15:68:D3:6B:B3
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:D5
Alias name: verisignclass2g2ca
MD5: 2D:BB:E5:25:D3:D1:65:82:3A:B7:0E:FA:E6:EB:E2:E1
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:A1
Alias name: mozillacert77.pem
MD5: CD:68:B6:A7:C7:C4:CE:75:E0:1D:4F:57:44:61:92:09
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:72:44
Alias name: mozillacert123.pem
MD5: C1:62:3E:23:C5:82:73:9C:03:59:4B:2B:E9:77:49:7F
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:69:27
Alias name: utndatacorpsgcca
MD5: B3:A5:3E:77:21:6D:AC:4A:C0:C9:FB:D5:41:3D:CA:06
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:B3:48
Alias name: mozillacert15.pem
MD5: 88:2C:8C:52:B8:A2:3C:F3:F7:BB:03:EA:AE:AC:42:0B
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:CB
Alias name: digicertglobalrootca
MD5: 79:E4:A9:84:0D:7D:3A:96:D7:C0:4F:E2:43:4C:89:2E
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:C7:01:61
Alias name: mozillacert66.pem
MD5: 3D:41:29:CB:1E:AA:11:74:CD:5D:B0:62:AF:B0:43:5B
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:D5:98
Alias name: mozillacert112.pem
MD5: 37:41:49:1B:18:56:9A:26:F5:AD:C2:66:FB:40:A5:4C
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:A3:89
Alias name: utnuserfirstclientauthemailca
MD5: D7:34:3D:EF:1D:27:09:28:E1:31:02:5B:13:2B:DD:F7
```

```
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:AE  
Alias name: verisignc2g1.pem  
MD5: B3:9C:25:B1:C3:2E:32:53:80:15:30:9D:4D:02:77:3E  
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:E9  
Alias name: mozillacert55.pem  
MD5: 74:9D:EA:60:24:C4:FD:22:53:3E:CC:3A:72:D9:29:4F  
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:15:57  
Alias name: mozillacert101.pem  
MD5: DF:F2:80:73:CC:F1:E6:61:73:FC:F5:42:E9:C5:7C:EE  
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:50  
Alias name: mozillacert119.pem  
MD5: 94:14:77:7E:3E:5E:FD:8F:30:BD:41:B0:CF:E7:D0:30  
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:9E  
Alias name: verisignc3g1.pem  
MD5: EF:5A:F1:33:EF:F1:CD:BB:51:02:EE:12:14:4B:96:C4  
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:14:03:05  
Alias name: mozillacert44.pem  
MD5: 72:E4:4A:87:E3:69:40:80:77:EA:BC:E3:F4:FF:F0:E1  
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:A3  
Alias name: mozillacert108.pem  
MD5: 3E:45:52:15:09:51:92:E1:B7:5D:37:9F:B1:87:29:8A  
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:99  
Alias name: mozillacert95.pem  
MD5: 3D:3B:18:9E:2C:64:5A:E8:D5:88:CE:0E:F9:37:C2:EC  
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:B2:4D  
Alias name: keynectisrootca  
MD5: CC:4D:AE:FB:30:6B:D8:38:FE:50:EB:86:61:4B:D2:26  
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:A3:32  
Alias name: mozillacert141.pem  
MD5: A9:23:75:9B:BA:49:36:6E:31:C2:DB:F2:E7:66:BA:87  
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:D6:93:55:59:6B  
Alias name: equifaxsecureglobalebusinessca1  
MD5: 51:F0:2A:33:F1:F5:55:39:07:F2:16:7A:47:C7:5D:63  
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:AE  
Alias name: affirmtrustpremiumca  
MD5: C4:5D:0E:48:B6:AC:28:30:4E:0A:BC:F9:38:16:87:57  
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:9A  
Alias name: baltimorecodesigningca  
MD5: 90:F5:28:49:56:D1:5D:2C:B0:53:D4:4B:EF:6F:90:22  
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:46:87  
Alias name: mozillacert33.pem  
MD5: 22:2D:A6:01:EA:7C:0A:F7:F0:6C:56:43:3F:77:76:D3  
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:37  
Alias name: mozillacert0.pem  
MD5: CA:3D:D3:68:F1:03:5C:D0:32:FA:B8:2B:59:E8:5A:DB  
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:36  
Alias name: mozillacert84.pem  
MD5: 49:63:AE:27:F4:D5:95:3D:D8:DB:24:86:B8:9C:07:53  
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:FB
```

```
Alias name: mozillacert130.pem
MD5: 65:58:AB:15:AD:57:6C:1E:A8:A7:B5:69:AC:BF:FF:EB
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:01:04
Alias name: mozillacert148.pem
MD5: 4C:56:41:E5:0D:BB:2B:E8:CA:A3:ED:18:08:AD:43:39
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:D8:81:21:37
Alias name: mozillacert22.pem
MD5: 02:26:C3:01:5E:08:30:37:43:A9:D0:7D:CF:37:E6:BF
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:D0:6C
Alias name: verisignc1g1.pem
MD5: 97:60:E8:57:5F:D3:50:47:E5:43:0C:94:36:8A:B0:62
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:33:55
Alias name: mozillacert7.pem
MD5: 32:4A:4B:BB:C8:63:69:9B:BE:74:9A:C6:DD:1D:46:24
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:B6:58
Alias name: mozillacert73.pem
MD5: D6:39:81:C6:52:7E:96:69:FC:FC:CA:66:ED:05:F2:96
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:F5
Alias name: mozillacert137.pem
MD5: D3:D9:BD:AE:9F:AC:67:24:B3:C8:1B:52:E1:B9:A9:BD
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:E8
Alias name: swisssignsilver2ca
MD5: E0:06:A1:C9:7D:CF:C9:FC:0D:CO:56:75:96:D8:62:13
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:D5
Alias name: mozillacert11.pem
MD5: 87:CE:0B:7B:2A:0E:49:00:E1:58:71:9B:37:A8:93:72
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:C6:32:5C
Alias name: mozillacert29.pem
MD5: D3:F3:A6:16:C0:FA:6B:1D:59:B1:2D:96:4D:0E:11:2E
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:83:86:0C
Alias name: mozillacert62.pem
MD5: EF:5A:F1:33:EF:F1:CD:BB:51:02:EE:12:14:4B:96:C4
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:14:03:05
Alias name: mozillacert126.pem
MD5: 77:0D:19:B1:21:FD:00:42:9C:3E:0C:A5:DD:0B:02:8E
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:4C
Alias name: securetrustca
MD5: DC:32:C3:A7:6D:25:57:C7:68:09:9D:EA:2D:A9:A2:D1
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:73
Alias name: soneraaclass1ca
MD5: 33:B7:84:F5:5F:27:D7:68:27:DE:14:DE:12:2A:ED:6F
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:32:31:3D
Alias name: mozillacert18.pem
MD5: F1:6A:22:18:C9:CD:DF:CE:82:1D:1D:B7:78:5C:A9:A5
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:A0
Alias name: mozillacert51.pem
MD5: 18:98:C0:D6:E9:3A:FC:F9:B0:F5:0C:F7:4B:01:44:17
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:BB
Alias name: mozillacert69.pem
```

```
MD5: A6:B0:CD:85:80:DA:5C:50:34:A3:39:90:2F:55:67:73
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:D4:53:1F
Alias name: mozillacert115.pem
MD5: 2B:9B:9E:E4:7B:6C:1F:00:72:1A:CC:C1:77:79:DF:6A
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:17:05:25:52
Alias name: verisignclass3g5ca
MD5: CB:17:E4:31:67:3E:E2:09:FE:45:57:93:F3:0A:FA:1C
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:DF
Alias name: utnuserfirsthardwareca
MD5: 4C:56:41:E5:0D:BB:2B:E8:CA:A3:ED:18:08:AD:43:39
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:D8:81:21:37
Alias name: addtrustqualifiedca
MD5: 27:EC:39:47:CD:DA:5A:AF:E2:9A:01:65:21:A9:4C:BB
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:D0:60:16
Alias name: mozillacert40.pem
MD5: 56:5F:AA:80:61:12:17:F6:67:21:E6:2B:6D:61:56:8E
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:B3:40:8E
Alias name: mozillacert58.pem
MD5: 01:5E:D8:6B:BD:6F:3D:8E:A1:31:F8:12:E0:98:73:6A
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:D7:66
Alias name: verisignclass3g3ca
MD5: CD:68:B6:A7:C7:C4:CE:75:E0:1D:4F:57:44:61:92:09
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:72:44
Alias name: mozillacert104.pem
MD5: 55:5D:63:00:97:BD:6A:97:F5:67:AB:4B:FB:6E:63:15
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:F7
Alias name: mozillacert91.pem
MD5: 30:C9:E7:1E:6B:E6:14:EB:65:B2:16:69:20:31:67:4D
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:55:7D
Alias name: thawtepersonalfreemailca
MD5: 53:4B:1D:17:58:58:1A:30:A1:90:F8:6E:5C:F2:CF:65
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:FA:4C:15:64:E1:84
Alias name: certplusclass3ppprimaryca
MD5: E1:4B:52:73:D7:1B:DB:93:30:E5:BD:E4:09:6E:BE:FB
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:02:85
Alias name: verisignc3g4.pem
MD5: 3A:52:E1:E7:FD:6F:3A:E3:6F:F3:6F:99:1B:F9:22:41
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:79
Alias name: swisssigngoldg2ca
MD5: 24:77:D9:A8:91:D1:3B:FA:88:2D:C2:FF:F8:CD:33:93
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:95
Alias name: mozillacert47.pem
MD5: ED:41:F5:8C:50:C5:2B:9C:73:E6:EE:6C:EB:C2:A8:26
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:C5:2A
Alias name: mozillacert80.pem
MD5: 64:B0:09:55:CF:B1:D5:99:E2:BE:13:AB:A6:5D:EA:4D
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:84:88:82:14:23
Alias name: mozillacert98.pem
MD5: 43:5E:88:D4:7D:1A:4A:7E:FD:84:2E:52:EB:01:D4:6F
```

```
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:76  
Alias name: mozillacert144.pem  
MD5: A3:EC:75:0F:2E:88:DF:FA:48:01:4E:0B:5C:48:6F:FB  
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:D9:27  
Alias name: starfieldclass2ca  
MD5: 32:4A:4B:BB:C8:63:69:9B:BE:74:9A:C6:DD:1D:46:24  
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:B6:58  
Alias name: mozillacert36.pem  
MD5: F0:96:B6:2F:C5:10:D5:67:8E:83:25:32:E8:5E:2E:E5  
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:D0  
Alias name: mozillacert3.pem  
MD5: 39:16:AA:B9:6A:41:E1:14:69:DF:9E:6C:3B:72:DC:B6  
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:03:12  
Alias name: globalsignr2ca  
MD5: 94:14:77:7E:3E:5E:FD:8F:30:BD:41:B0:CF:E7:D0:30  
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:9E  
Alias name: mozillacert87.pem  
MD5: 6C:39:7D:A4:0E:55:59:B2:3F:D6:41:B1:12:50:DE:43  
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:F6  
Alias name: mozillacert133.pem  
MD5: D6:ED:3C:CA:E2:66:0F:AF:10:43:0D:77:9B:04:09:BF  
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:BD  
Alias name: mozillacert25.pem  
MD5: CB:17:E4:31:67:3E:E2:09:FE:45:57:93:F3:0A:FA:1C  
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:DF  
Alias name: verisignclass1g2ca  
MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83  
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:7F  
Alias name: mozillacert76.pem  
MD5: 82:92:BA:5B:EF:CD:8A:6F:A6:3D:55:F9:84:F6:D6: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:A7  
Alias name: mozillacert122.pem  
MD5: 1D:35:54:04:85:78:B0:3F:42:42:4D:BF:20:73:0A:3F  
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:F2  
Alias name: mozillacert14.pem  
MD5: D4:74:DE:57:5C:39:B2:D3:9C:85:83:C5:C0:65:49:8A  
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:CF  
Alias name: equifaxsecureca  
MD5: 67:CB:9D:C0:13:24:8A:82:9B:B2:17:1E:D1:1B:EC:D4  
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:F1:78  
Alias name: mozillacert65.pem  
MD5: A2:6F:53:B7:EE:40:DB:4A:68:E7:FA:18:D9:10:4B:72  
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:73:01:9A  
Alias name: mozillacert111.pem  
MD5: F9:03:7E:CF:E6:9E:3C:73:7A:2A:90:07:69:FF:2B:96  
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:1B  
Alias name: certumtrustednetworkca  
MD5: D5:E9:81:40:C5:18:69:FC:46:2C:89:75:62:0F:AA:78  
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:C4:40:8E
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
Alias name: mozillacert129.pem
MD5: 92:65:58:8B:A2:1A:31:72:73:68:5C:B4:A5:7A:07:48
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:B9:12
Alias name: mozillacert54.pem
MD5: B5:E8:34:36:C9:10:44:58:48:70:6D:2E:83:D4:B8:05
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:D4
Alias name: mozillacert100.pem
MD5: CD:E0:25:69:8D:47:AC:9C:89:35:90:F7:FD:51:3D:2F
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:C1
Alias name: mozillacert118.pem
MD5: 8F:5D:77:06:27:C4:98:3C:5B:93:78:E7:D7:7D:9B:CC
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:07
Alias name: gd-class2-root.pem
MD5: 91:DE:06:25:AB:DA:FD:32:17:0C:BB:25:17:2A:84:67
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:E4
Alias name: mozillacert151.pem
MD5: 86:38:6D:5E:49:63:6C:85:5C:DB:6D:DC:94:B7:D0:F7
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:F6:67
Alias name: thawteprimaryrootcag3
MD5: FB:1B:5D:43:8A:94:CD:44:C6:76:F2:43:4B:47:E7:31
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:D3:4C
Alias name: quovadisrootca
MD5: 27:DE:36:FE:72:B7:00:03:00:9D:F4:F0:1E:6C:04:24
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:88:73
Alias name: thawteprimaryrootcag2
MD5: 74:9D:EA:60:24:C4:FD:22:53:3E:CC:3A:72:D9:29:4F
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:15:57
Alias name: deprecateditsecca
MD5: A5:96:0C:F6:B5:AB:27:E5:01:C6:00:88:9E:60:33:E5
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:CB
Alias name: entrustrootcag2
MD5: 4B:E2:C9:91:96:65:0C:F4:0E:5A:93:92:A0:0A:FE:B2
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:F3:39
Alias name: mozillacert43.pem
MD5: 40:01:25:06:8D:21:43:6A:0E:43:00:9C:E7:43:F3:D5
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:D2
Alias name: mozillacert107.pem
MD5: BE:EC:11:93:9A:F5:69:21:BC:D7:C1:C0:67:89:CC:2A
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:CE
Alias name: trustcenterclass4caii
MD5: 9D:FB:F9:AC:ED:89:33:22:F4:28:48:83:25:23:5B:E0
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:32
Alias name: mozillacert94.pem
MD5: 46:A7:D2:FE:45:FB:64:5A:A8:59:90:9B:78:44:9B:29
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:48
Alias name: mozillacert140.pem
MD5: 5E:39:7B:DD:F8:BA:EC:82:E9:AC:62:BA:0C:54:00:2B
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:86
Alias name: ttelesecglobalrootclass3ca
MD5: CA:FB:40:A8:4E:39:92:8A:1D:FE:8E:2F:C4:27:EA:EF
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
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:BD  
Alias name: amzninternalcorpca  
MD5: 7B:0E:9D:67:A9:3A:88:DD:BA:81:8D:A9:3C:74:AA:BB  
SHA256:  
01:29:04:6C:60:EF:5C:51:60:D3:9F:A2:3A:1D:0C:52:0A:AF:DA:4F:17:87:95:AA:66:82:01:9F:76:C9:11:DC  
Alias name: starfieldservicesrootg2ca  
MD5: 17:35:74:AF:7B:61:1C:EB:F4:F9:3C:E2:EE:40:F9:A2  
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:B5  
Alias name: mozillacert32.pem  
MD5: 0C:7F:DD:6A:F4:2A:B9:C8:9B:BD:20:7E:A9:DB:5C:37  
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:89:32  
Alias name: mozillacert83.pem  
MD5: 2C:8C:17:5E:B1:54:AB:93:17:B5:36:5A:DB:D1:C6:F2  
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:63:0E  
Alias name: verisignroot.pem  
MD5: 8E:AD:B5:01:AA:4D:81:E4:8C:1D:D1:E1:14:00:95:19  
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:3C  
Alias name: mozillacert147.pem  
MD5: B3:A5:3E:77:21:6D:AC:4A:C0:C9:FB:D5:41:3D:CA:06  
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:B3:48  
Alias name: camerfirmachambersca  
MD5: 5E:80:9E:84:5A:0E:65:0B:17:02:F3:55:18:2A:3E:D7  
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:C0  
Alias name: mozillacert21.pem  
MD5: E0:06:A1:C9:7D:CF:C9:FC:OD:C0:56:75:96:D8:62:13  
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:D5  
Alias name: mozillacert39.pem  
MD5: CE:78:33:5C:59:78:01:6E:18:EA:B9:36:A0:B9:2E:23  
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:B4  
Alias name: mozillacert6.pem  
MD5: 91:DE:06:25:AB:DA:FD:32:17:0C:BB:25:17:2A:84:67  
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:E4  
Alias name: verisignuniversalrootca  
MD5: 8E:AD:B5:01:AA:4D:81:E4:8C:1D:D1:E1:14:00:95:19  
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:3C  
Alias name: mozillacert72.pem  
MD5: 80:3A:BC:22:C1:E6:FB:8D:9B:3B:27:4A:32:1B:9A:01  
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:DA  
Alias name: geotrustuniversalca  
MD5: 92:65:58:8B:A2:1A:31:72:73:68:5C:B4:A5:7A:07:48  
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:B9:12  
Alias name: mozillacert136.pem  
MD5: 49:79:04:B0:EB:87:19:AC:47:BO:BC:11:51:9B:74:D0  
SHA256:  
D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:OC:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F4  
Alias name: mozillacert10.pem  
MD5: F8:38:7C:77:88:DF:2C:16:68:2E:C2:E2:52:4B:B8:F9  
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:D9:2E  
Alias name: mozillacert28.pem  
MD5: 5C:48:DC:F7:42:72:EC:56:94:6D:1C:CC:71:35:80:75  
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:66
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
Alias name: affirmtrustnetworkingca
MD5: 42:65:CA:BE:01:9A:9A:4C:A9:8C:41:49:CD:C0:D5:7F
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:B4:1B
Alias name: mozillacert61.pem
MD5: 42:81:A0:E2:1C:E3:55:10:DE:55:89:42:65:96:22:E6
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:43:4A
Alias name: mozillacert79.pem
MD5: C4:5D:0E:48:B6:AC:28:30:4E:0A:BC:F9:38:16:87:57
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:9A
Alias name: affirmtrustcommercialca
MD5: 82:92:BA:5B:EF:CD:8A:6F:A6:3D:55:F9:84:F6:D6: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:A7
Alias name: mozillacert125.pem
MD5: D6:A5:C3:ED:5D:DD:3E:00:C1:3D:87:92:1F:1D:3F:E4
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:4C
Alias name: mozillacert17.pem
MD5: 21:D8:4C:82:2B:99:09:33:A2:EB:14:24:8D:8E:5F:E8
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:40
Alias name: mozillacert50.pem
MD5: 2C:20:26:9D:CB:1A:4A:00:85:B5:B7:5A:AE:C2:01:37
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:B2
Alias name: mozillacert68.pem
MD5: 73:3A:74:7A:EC:BB:A3:96:A6:C2:E4:E2:C8:9B:C0:C3
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:EF
Alias name: starfieldrootg2ca
MD5: D6:39:81:C6:52:7E:96:69:FC:FC:CA:66:ED:05:F2:96
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:F5
Alias name: mozillacert114.pem
MD5: B8:A1:03:63:B0:BD:21:71:70:8A:6F:13:3A:BB:79:49
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:F2:3C
Alias name: buypassclass3ca
MD5: 3D:3B:18:9E:2C:64:5A:E8:D5:88:CE:0E:F9:37:C2:EC
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:B2:4D
Alias name: mozillacert57.pem
MD5: A8:0D:6F:39:78:B9:43:6D:77:42:6D:98:5A:CC:23:CA
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:B2
Alias name: verisignc2g3.pem
MD5: F8:BE:C4:63:22:C9:A8:46:74:8B:B8:1D:1E:4A:2B:F6
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:BB
Alias name: verisignclass2g3ca
MD5: F8:BE:C4:63:22:C9:A8:46:74:8B:B8:1D:1E:4A:2B:F6
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:BB
Alias name: mozillacert103.pem
MD5: E6:24:E9:12:01:AE:0C:DE:8E:85:C4:CE:A3:12:DD:EC
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:B0
Alias name: mozillacert90.pem
MD5: 69:C1:0D:4F:07:A3:1B:C3:FE:56:3D:04:BC:11:F6:A6
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:D7:60:66
Alias name: verisignc3g3.pem
MD5: CD:68:B6:A7:C7:C4:CE:75:E0:1D:4F:57:44:61:92:09
```

```
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:72:44  
Alias name: mozillacert46.pem  
MD5: AA:8E:5D:D9:F8:DB:0A:58:B7:8D:26:87:6C:82:35:55  
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:E9:39  
Alias name: godaddyclass2ca  
MD5: 91:DE:06:25:AB:DA:FD:32:17:0C:BB:25:17:2A:84:67  
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:E4  
Alias name: verisignc4g3.pem  
MD5: DB:C8:F2:27:2E:B1:EA:6A:29:23:5D:FE:56:3E:33:DF  
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:96:05:61:79:15:06  
Alias name: mozillacert97.pem  
MD5: A2:33:9B:4C:74:78:73:D4:6C:E7:C1:F3:8D:CB:5C:E9  
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:A8:88:63:8B  
Alias name: mozillacert143.pem  
MD5: F1:BC:63:6A:54:E0:B5:27:F5:CD:E7:1A:E3:4D:6E:4A  
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:6C  
Alias name: mozillacert35.pem  
MD5: 3F:45:96:39:E2:50:87:F7:BB:FE:98:0C:3C:20:98:E6  
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:CF  
Alias name: mozillacert2.pem  
MD5: 3A:52:E1:E7:FD:6F:3A:E3:6F:F3:6F:99:1B:F9:22:41  
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:79  
Alias name: utnuserfirstobjectca  
MD5: A7:F2:E4:16:06:41:11:50:30:6B:9C:E3:B4:9C:B0:C9  
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:E5:8F  
Alias name: mozillacert86.pem  
MD5: 10:FC:63:5D:F6:26:3E:0D:F3:25:BE:5F:79:CD:67:67  
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:70  
Alias name: mozillacert132.pem  
MD5: 14:F1:08:AD:9D:FA:64:E2:89:E7:1C:CF:A8:AD:7D:5E  
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:E3  
Alias name: addtrustclass1ca  
MD5: 1E:42:95:02:33:92:6B:B9:5F:CO:7F:DA:D6:B2:4B:FC  
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:A7  
Alias name: mozillacert24.pem  
MD5: 7C:A5:0F:F8:5B:9A:7D:6D:30:AE:54:5A:E3:42:A2:8A  
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:B6:71:6F  
Alias name: verisignc1g3.pem  
MD5: B1:47:BC:18:57:D1:18:A0:78:2D:EC:71:E8:2A:95:73  
SHA256:  
CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:CO:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:65  
Alias name: mozillacert9.pem  
MD5: 37:85:44:53:32:45:1F:20:F0:F3:95:E1:25:C4:43:4E  
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:B3  
Alias name: amzninternalrootca  
MD5: 08:09:73:AC:E0:78:41:7C:0A:26:33:51:E8:CF:E6:60  
SHA256:  
OE: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:AA  
Alias name: mozillacert75.pem  
MD5: 67:CB:9D:C0:13:24:8A:82:9B:B2:17:1E:D1:1B:EC:D4  
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:F1:78
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
Alias name: entrustevca
MD5: D6:A5:C3:ED:5D:DD:3E:00:C1:3D:87:92:1F:1D:3F:E4
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:4C
Alias name: secomscrootca2
MD5: 6C:39:7D:A4:0E:55:59:B2:3F:D6:41:B1:12:50:DE:43
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:F6
Alias name: camerfirmachambersignca
MD5: 9E:80:FF:78:01:0C:2E:C1:36:BD:FE:96:90:6E:08:F3
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:CA
Alias name: secomscrootca1
MD5: F1:BC:63:6A:54:EC:B5:27:F5:CD:E7:1A:E3:4D:6E:4A
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:6C
Alias name: mozillacert121.pem
MD5: 1E:42:95:02:33:92:6B:B9:5F:CO:7F:DA:D6:B2:4B:FC
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:A7
Alias name: mozillacert139.pem
MD5: 27:DE:36:FE:72:B7:00:03:00:9D:F4:F0:1E:6C:04:24
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:88:73
Alias name: mozillacert13.pem
MD5: C5:A1:B7:FF:73:DD:D6:D7:34:32:18:DF:FC:3C:AD:88
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:98
Alias name: mozillacert64.pem
MD5: 06:9F:69:79:16:66:90:02:1B:8C:A2:C3:07:6F:3A
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:72
Alias name: mozillacert110.pem
MD5: D0:A0:5A:EE:05:B6:09:94:21:A1:7D:F1:B2:29:82:02
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:D1:04:38:41:13
Alias name: mozillacert128.pem
MD5: 0E:40:A7:6C:DE:03:5D:8F:D1:0F:E4:D1:8D:F9:6C:A9
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:85
Alias name: entrust2048ca
MD5: EE:29:31:BC:32:7E:9A:E6:E8:B5:F7:51:B4:34:71:90
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:B1:77
Alias name: mozillacert53.pem
MD5: 7E:23:4E:5B:A7:A5:B4:25:E9:00:07:74:11:62:AE:D6
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:19
Alias name: mozillacert117.pem
MD5: AC:B6:94:A5:9C:17:E0:D7:91:52:9B:B1:97:06:A6:E4
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:EB
Alias name: mozillacert150.pem
MD5: C5:E6:7B:BF:06:D0:4F:43:ED:C4:7A:65:8A:FB:6B:19
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:ED
Alias name: thawteserverca
MD5: EE:FE:61:69:65:6E:F8:9C:C6:2A:F4:D7:2B:63:EF:A2
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:10:55:31:6B
Alias name: secomvalicertclass1ca
MD5: 65:58:AB:15:AD:57:6C:1E:A8:A7:B5:69:AC:BF:FF:EB
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:01:04
Alias name: mozillacert42.pem
MD5: 74:01:4A:91:B1:08:C4:58:CE:47:CD:F0:DD:11:53:08
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
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:D3  
Alias name: verisignc2g6.pem  
MD5: 7D:0B:83:E5:FB:7C:AD:07:4F:20:A9:B5:DF:63:ED:79  
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:F0  
Alias name: godaddyroot2ca  
MD5: 80:3A:BC:22:C1:E6:FB:8D:9B:3B:27:4A:32:1B:9A:01  
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:DA  
Alias name: gtecybertrustglobalca  
MD5: CA:3D:D3:68:F1:03:5C:D0:32:FA:B8:2B:59:E8:5A:DB  
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:36  
Alias name: mozillacert106.pem  
MD5: 7B:30:34:9F:DD:0A:4B:6B:35:CA:31:51:28:5D:AE:EC  
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:B2:58:01:5D  
Alias name: equifaxsecurebusinessca1  
MD5: 14:CO:08:E5:A3:85:03:A3:BE:78:E9:67:4F:27:CA:EE  
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:F5:96  
Alias name: mozillacert93.pem  
MD5: 78:4B:FB:9E:64:82:0A:D3:B8:4C:62:F3:64:F2:90:64  
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:59:95  
Alias name: quovaldisrootca3  
MD5: 31:85:3C:62:94:97:63:B9:AA:FD:89:4E:AF:6F:E0:CF  
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:35  
Alias name: quovaldisrootca2  
MD5: 5E:39:7B:DD:F8:BA:EC:82:E9:AC:62:BA:0C:54:00:2B  
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:86  
Alias name: soneraaclass2ca  
MD5: A3:EC:75:0F:2E:88:DF:FA:48:01:4E:0B:5C:48:6F:FB  
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:D9:27  
Alias name: mozillacert31.pem  
MD5: 7C:62:FF:74:9D:31:53:5E:68:4A:D5:78:AA:1E:BF:23  
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:C7  
Alias name: mozillacert49.pem  
MD5: DF:3C:73:59:81:E7:39:50:81:04:4C:34:A2:CB:B3:7B  
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:24  
Alias name: mozillacert82.pem  
MD5: 7F:30:78:8C:03:E3:CA:C9:0A:E2:C9:EA:1E:AA:55:1A  
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:17  
Alias name: mozillacert146.pem  
MD5: 91:F4:03:55:20:A1:F8:63:2C:62:DE:AC:FB:61:1C:8E  
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:E7  
Alias name: baltimorecybertrustca  
MD5: AC:B6:94:A5:9C:17:E0:D7:91:52:9B:B1:97:06:A6:E4  
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:EB  
Alias name: mozillacert20.pem  
MD5: 24:77:D9:A8:91:D1:3B:FA:88:2D:C2:FF:F8:CD:33:93  
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:95  
Alias name: mozillacert38.pem  
MD5: 93:2A:3E:F6:FD:23:69:0D:71:20:D4:2B:47:99:2B:A6  
SHA256:  
A6:C5:1E:0D:A5:CA:0A:93:09:D2:E4:C0:E4:OC:2A:F9:10:7A:AE:82:03:85:7F:E1:98:E3:E7:69:E3:43:08:5C
```

Amazon API Gateway Developer Guide
Supported Certificate Authorities for
HTTP and HTTP Proxy Integration

```
Alias name: mozillacert5.pem
MD5: A1:0B:44:B3:CA:10:D8:00:6E:9D:0F:D8:0F:92:0A:D1
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:A2
Alias name: mozillacert71.pem
MD5: 9E:80:FF:78:01:0C:2E:C1:36:BD:FE:96:90:6E:08:F3
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:CA
Alias name: verisignclass3g4ca
MD5: 3A:52:E1:E7:FD:6F:3A:E3:6F:F3:6F:99:1B:F9:22:41
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:79
Alias name: mozillacert89.pem
MD5: DB:C8:F2:27:2E:B1:EA:6A:29:23:5D:FE:56:3E:33:DF
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:96:05:61:79:15:06
Alias name: mozillacert135.pem
MD5: 2C:8F:9F:66:1D:18:90:B1:47:26:9D:8E:86:82:8C:A9
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:D8:14:36:24
Alias name: camerfirmachamberscommerceca
MD5: B0:01:EE:14:D9:AF:29:18:94:76:8E:F1:69:33:2A:84
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:C3
Alias name: mozillacert27.pem
MD5: CF:F4:27:0D:D4:ED:DC:65:16:49:6D:3D:DA:BF:6E:DE
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:69
Alias name: verisignc1g6.pem
MD5: 2F:A8:B4:DA:F6:64:4B:1E:82:F9:46:3D:54:1A:7C:B0
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:B9
Alias name: verisignclass3g2ca
MD5: A2:33:9B:4C:74:78:73:D4:6C:E7:C1:F3:8D:CB:5C:E9
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:A8:88:63:8B
Alias name: mozillacert60.pem
MD5: B7:52:74:E2:92:B4:80:93:F2:75:E4:CC:D7:F2:EA:26
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:42:16:12
Alias name: mozillacert78.pem
MD5: 42:65:CA:BE:01:9A:9A:4C:A9:8C:41:49:CD:C0:D5:7F
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:B4:1B
Alias name: gd_bundle-g2.pem
MD5: 96:C2:50:31:BC:0D:C3:5C:FB:A7:23:73:1E:1B:41:40
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:A6
Alias name: certumca
MD5: 2C:8F:9F:66:1D:18:90:B1:47:26:9D:8E:86:82:8C:A9
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:D8:14:36:24
Alias name: deutschetelekomrootca2
MD5: 74:01:4A:91:B1:08:C4:58:CE:47:CD:F0:DD:11:53:08
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:D3
Alias name: mozillacert124.pem
MD5: 27:EC:39:47:CD:DA:5A:AF:E2:9A:01:65:21:A9:4C:BB
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:D0:60:16
Alias name: mozillacert16.pem
MD5: 41:03:52:DC:0F:F7:50:1B:16:F0:02:8E:BA:6F:45:C5
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:D6:77:07:39
Alias name: secomevrootca1
MD5: 22:2D:A6:01:EA:7C:0A:F7:F0:6C:56:43:3F:77:76:D3
```

```
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:37  
Alias name: mozillacert67.pem  
MD5: C5:DF:B8:49:CA:05:13:55:EE:2D:BA:1A:C3:3E:B0:28  
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:C5:3B  
Alias name: globalsignr3ca  
MD5: C5:DF:B8:49:CA:05:13:55:EE:2D:BA:1A:C3:3E:B0:28  
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:C5:3B  
Alias name: mozillacert113.pem  
MD5: EE:29:31:BC:32:7E:9A:E6:E8:B5:F7:51:B4:34:71:90  
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:B1:77  
Alias name: gdroot-g2.pem  
MD5: 80:3A:BC:22:C1:E6:FB:8D:9B:3B:27:4A:32:1B:9A:01  
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:DA  
Alias name: aolrootca2  
MD5: D6:ED:3C:CA:E2:66:0F:AF:10:43:0D:77:9B:04:09:BF  
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:BD  
Alias name: trustcenteruniversalcai  
MD5: 45:E1:A5:72:C5:A9:36:64:40:9E:F5:E4:58:84:67:8C  
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:E7  
Alias name: aolrootca1  
MD5: 14:F1:08:AD:9D:FA:64:E2:89:E7:1C:CF:A8:AD:7D:5E  
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:E3  
Alias name: verisignc2g2.pem  
MD5: 2D:BB:E5:25:D3:D1:65:82:3A:B7:0E:FA:E6:EB:E2:E1  
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:A1  
Alias name: mozillacert56.pem  
MD5: FB:1B:5D:43:8A:94:CD:44:C6:76:F2:43:4B:47:E7:31  
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:D3:4C  
Alias name: verisignclass1g3ca  
MD5: B1:47:BC:18:57:D1:18:A0:78:2D:EC:71:E8:2A:95:73  
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:65  
Alias name: mozillacert102.pem  
MD5: AA:C6:43:2C:5E:2D:CD:C4:34:C0:50:4F:11:02:4F:B6  
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:D3:48:81  
Alias name: addtrustexternalalca  
MD5: 1D:35:54:04:85:78:B0:3F:42:42:4D:BF:20:73:0A:3F  
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:F2  
Alias name: verisignc3g2.pem  
MD5: A2:33:9B:4C:74:78:73:D4:6C:E7:C1:F3:8D:CB:5C:E9  
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:A8:88:63:8B  
Alias name: verisignclass3ca  
MD5: EF:5A:F1:33:EF:F1:CD:BB:51:02:EE:12:14:4B:96:C4  
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:14:03:05  
Alias name: mozillacert45.pem  
MD5: 1B:2E:00:CA:26:06:90:3D:AD:FE:6F:15:68:D3:6B:B3  
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:D5  
Alias name: verisignc4g2.pem  
MD5: 26:6D:2C:19:98:B6:70:68:38:50:54:19:EC:90:34:60  
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:C7:01:19:61
```

```
Alias name: digicertassuredidrootca
MD5: 87:CE:0B:7B:2A:0E:49:00:E1:58:71:9B:37:A8:93:72
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:C6:32:5C
Alias name: verisignclassica
MD5: 86:AC:DE:2B:C5:6D:C3:D9:8C:28:88:D3:8D:16:13:1E
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:B8:84:2C
Alias name: mozillacert109.pem
MD5: 26:01:FB:D8:27:A7:17:9A:45:54:38:1A:43:01:3B:03
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:F8:07:03
Alias name: thawtepremiumserverca
MD5: A6:6B:60:90:23:9B:3F:2D:BB:98:6F:D6:A7:19:0D:46
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:E9
Alias name: verisigntsaca
MD5: F2:89:95:6E:4D:05:F0:F1:A7:21:55:7D:46:11:BA:47
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:9D
Alias name: mozillacert96.pem
MD5: CA:FB:40:A8:4E:39:92:8A:1D:FE:8E:2F:C4:27:EA:EF
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:BD
Alias name: mozillacert142.pem
MD5: 31:85:3C:62:94:97:63:B9:AA:FD:89:4E:AF:6F:E0:CF
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:35
Alias name: thawteprimaryrootca
MD5: 8C:CA:DC:0B:22:CE:F5:BE:72:AC:41:1A:11:A8:D8:12
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:9F
Alias name: mozillacert34.pem
MD5: BC:6C:51:33:A7:E9:D3:66:63:54:15:72:1B:21:92:93
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:F5
Alias name: mozillacert1.pem
MD5: C5:70:C4:A2:ED:53:78:0C:C8:10:53:81:64:CB:D0:1D
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:E9
Alias name: xrampglobalca
MD5: A1:0B:44:B3:CA:10:D8:00:6E:9D:0F:D8:0F:92:0A:D1
SHA256:
CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:CO:FF:0B:CF:0D:32:86:FC:1A:A2
Alias name: mozillacert85.pem
MD5: AA:08:8F:F6:F9:7B:B7:F2:B1:A7:1E:9B:EA:EA:BD:79
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:44
Alias name: valicertclass2ca
MD5: A9:23:75:9B:BA:49:36:6E:31:C2:DB:F2:E7:66:BA:87
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:D6:93:55:59:6B
Alias name: mozillacert131.pem
MD5: 34:FC:B8:D0:36:DB:9E:14:B3:C2:F2:DB:8F:E4:94:C7
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:0B
Alias name: mozillacert149.pem
MD5: B0:01:EE:14:D9:AF:29:18:94:76:8E:F1:69:33:2A:84
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:C3
Alias name: geotrustprimaryca
MD5: 02:26:C3:01:5E:08:30:37:43:A9:D0:7D:CF:37:E6:BF
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:D0:6C
Alias name: mozillacert23.pem
MD5: 8C:CA:DC:0B:22:CE:F5:BE:72:AC:41:1A:11:A8:D8:12
```

```
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:9F  
Alias name: verisignc1g2.pem  
MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83  
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:7F  
Alias name: mozillacert8.pem  
MD5: 22:4D:8F:8A:FC:F7:35:C2:BB:57:34:90:7B:8B:22:16  
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:EA  
Alias name: mozillacert74.pem  
MD5: 17:35:74:AF:7B:61:1C:EB:F4:F9:3C:E2:EE:40:F9:A2  
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:B5  
Alias name: mozillacert120.pem  
MD5: 64:9C:EF:2E:44:FC:C6:8F:52:07:D0:51:73:8F:CB:3D  
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:F6:73  
Alias name: geotrustglobalca  
MD5: F7:75:AB:29:FB:51:4E:B7:77:5E:FF:05:3C:99:8E:F5  
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:A7:3A  
Alias name: mozillacert138.pem  
MD5: 91:1B:3F:6E:CD:9E:AB:EE:07:FE:1F:71:D2:B3:61:27  
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:69  
Alias name: mozillacert12.pem  
MD5: 79:E4:A9:84:0D:7D:3A:96:D7:CO:4F:E2:43:4C:89:2E  
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:C7:01:61  
Alias name: comodoaaaca  
MD5: 49:79:04:B0:EB:87:19:AC:47:B0:BC:11:51:9B:74:D0  
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:F4  
Alias name: mozillacert63.pem  
MD5: F8:49:F4:03:BC:44:2D:83:BE:48:69:7D:29:64:FC:B1  
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:F3:78  
Alias name: certplusclass2primaryca  
MD5: 88:2C:8C:52:B8:A2:3C:F3:F7:BB:03:EA:AE:AC:42:0B  
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:CB  
Alias name: mozillacert127.pem  
MD5: F7:75:AB:29:FB:51:4E:B7:77:5E:FF:05:3C:99:8E:F5  
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:A7:3A  
Alias name: ttelesecglobalrootclass2ca  
MD5: 2B:9B:9E:E4:7B:6C:1F:00:72:1A:CC:C1:77:79:DF:6A  
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:17:05:25:52  
Alias name: mozillacert19.pem  
MD5: 37:A5:6E:D4:B1:25:84:97:B7:FD:56:15:7A:F9:A2:00  
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:E6  
Alias name: digicerthighassuranceevrootca  
MD5: D4:74:DE:57:5C:39:B2:D3:9C:85:83:C5:C0:65:49:8A  
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:CF  
Alias name: amzninternalinfoseccag3  
MD5: E9:34:94:02:BA:BB:31:6B:22:E6:2B:A9:C4:F0:26:04  
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:68  
Alias name: mozillacert52.pem  
MD5: 21:BC:82:AB:49:C4:13:3B:4B:B2:2B:5C:6B:90:9C:19  
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:A7
```

```

Alias name: mozillacert116.pem
MD5: AE:B9:C4:32:4B:AC:7F:5D:66:CC:77:94:BB:2A:77:56
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:A4:74
Alias name: globalsignca
MD5: 3E:45:52:15:09:51:92:E1:B7:5D:37:9F:B1:87:29:8A
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:99
Alias name: mozillacert41.pem
MD5: 45:E1:A5:72:C5:A9:36:64:40:9E:F5:E4:58:84:67:8C
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:E7
Alias name: mozillacert59.pem
MD5: 8E:AD:B5:01:AA:4D:81:E4:8C:1D:D1:E1:14:00:95:19
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:3C
Alias name: mozillacert105.pem
MD5: 5B:04:69:EC:A5:83:94:63:18:A7:86:D0:E4:F2:6E:19
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:38:41
Alias name: trustcenterclass2caii
MD5: CE:78:33:5C:59:78:01:6E:18:EA:B9:36:A0:B9:2E:23
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:B4
Alias name: mozillacert92.pem
MD5: C9:3B:0D:84:41:FC:A4:76:79:23:08:57:DE:10:19:16
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:A9:11
Alias name: verisignc3g5.pem
MD5: CB:17:E4:31:67:3E:E2:09:FE:45:57:93:F3:0A:FA:1C
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:DF
Alias name: geotrustprimarycag3
MD5: B5:E8:34:36:C9:10:44:58:48:70:6D:2E:83:D4:B8:05
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:D4
Alias name: geotrustprimarycag2
MD5: 01:5E:D8:6B:BD:6F:3D:8E:A1:31:F8:12:E0:98:73:6A
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:D7:66
Alias name: mozillacert30.pem
MD5: 15:AC:A5:C2:92:2D:79:BC:E8:7F:CB:67:ED:02:CF:36
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:43
Alias name: affirmtrustpremiumeccca
MD5: 64:B0:09:55:CF:B1:D5:99:E2:BE:13:AB:A6:5D:EA:4D
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:84:88:82:14:23
Alias name: mozillacert48.pem
MD5: B8:08:9A:F0:03:CC:1B:0D:C8:6C:0B:76:A1:75:64:23
SHA256:
OF: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:38

```

Create and Use API Gateway Usage Plans

After you create, test, and deploy your APIs, you can use API Gateway usage plans to extend them as product offerings for your customers. You can provide usage plans to allow specified customers to access selected APIs at agreed-upon request rates and quotas that can meet their business requirements and budget constraints.

What Is a Usage Plan?

A usage plan prescribes who can access one or more deployed API stages—and also how much and how fast the caller can access the APIs. The plan uses an API key to identify an API client and meters access to an API stage with the configurable throttling and quota limits that are enforced on individual client API keys.

The throttling prescribes the request rate limits that are applied to each API key. The quotas are the maximum number of requests with a given API key submitted within a specified time interval. You can configure individual API methods to require API key authorization based on usage plan configuration. An API stage is identified by an API identifier and a stage name.

Note

Throttling and quota limits apply to requests for individual API keys that are aggregated across all API stages within a usage plan.

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" \(p. 317\)](#).

Expose API Key Source

When you enact a usage plan for an API and enable API keys on API methods, the incoming request must come with an appropriate API key. API Gateway reads this API key from an incoming request and verifies it against the key in the usage plan. If the keys match, API Gateway throttles the requests according to the plan's request limit and quota. Otherwise, it throws an `InvalidKeyParameter` exception. As a result, the caller receives a `403 Forbidden` response.

There are two sources that API Gateway receives an API key from. You can distribute the key to client developers and ask the client to pass the API key as the `X-API-Key` header of an incoming request when you require API keys on API methods. This is known as the `HEADER` source because API Gateway reads it from the header. Alternatively, you can 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" \(p. 279\)](#). This is known as the `AUTHORIZER` source because API Gateway reads it from the authorizer output.

To choose an API key source using the API Gateway console, follow the instructions below:

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 primary navigation pane, choose **Settings** under the chosen or newly created API.
4. Under the **API Key Source** section in the **Settings** pane, choose `HEADER` or `AUTHORIZER` from the drop-down list.
5. 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 above 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/fugvwdxtri/ 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 API key source type you choose, use one of the following procedures to use a header-sourced API key or an authorizedReturned API key for a method invocation:

To use a header-sourced API key for method invocation:

1. Create an API with desired API methods. And 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 an authorized-sourced API key for method invocation:

1. Create an API with desired API methods. And 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 custom Lambda authorizer of the token type. Include, as a root-level property of the authorization response, `usageIdentifierKey:{api-key}`, where `{api-key}` stands for the API key value mentioned in the previous step.
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 authorizerReturned API key is used automatically.

How to Configure a Usage Plan?

The following steps outline how you, as the API owner, 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 in stages.
2. Generate API keys and distribute the keys to app developers (your customers) by using your APIs.
3. Create the usage plan with the desired throttle and quota limits.
4. Associate selected API stages and API keys to 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 \(p. 317\)](#). For user authentication and authorization, don't use API keys. Use an IAM role, a Lambda authorizer (p. 272), or an [Amazon Cognito user pool \(p. 286\)](#).

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.

For instructions on how to create and deploy an API by using the API Gateway console, see [Creating an API in Amazon API Gateway \(p. 81\)](#) and [Deploying an API in Amazon API Gateway \(p. 370\)](#), respectively.

Topics

- [Require API Key on a Method \(p. 317\)](#)
- [Create an API Key \(p. 318\)](#)
- [Import API Keys \(p. 319\)](#)

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 AWS Management Console and open the API Gateway console at <https://console.aws.amazon.com/apigateway/>.
2. In the API Gateway main navigation pane, choose **Resources**.
3. Under **Resources**, create a new method or choose an existing one.
4. Choose **Method Request**.
5. Under the **Authorization Settings** section, choose `true` for **API Key Required**.
6. Select the checkmark icon to save the settings.

The screenshot shows the 'Method Execution' section for a GET request on the '/pets' endpoint. The 'Authorization' dropdown is set to 'NONE'. The 'API Key Required' dropdown is set to 'true' and is highlighted with a red circle. Other sections visible include 'URL Query String Parameters', 'HTTP Request Headers', and 'Request Models'.

7. 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 AWS Management Console and open the API Gateway console at <https://console.aws.amazon.com/apigateway/>.
2. In the API Gateway main navigation pane, choose **API Keys**.
3. From the **Actions** drop-down menu, choose **Create API key**.

The screenshot shows the 'API Keys' page. The 'Actions' dropdown menu is open, showing 'Create API key' and 'Import API keys'. The 'API Keys' section is selected in the left sidebar.

4. In **Create API Key**, do the following:
 - a. Type an API key name (for example, **MyFirstKey**) in the **Name** input field.
 - b. Choose **Auto Generate** to have API Gateway generate the key value, or choose **Custom** to enter the key manually.

- c. Choose **Save**.

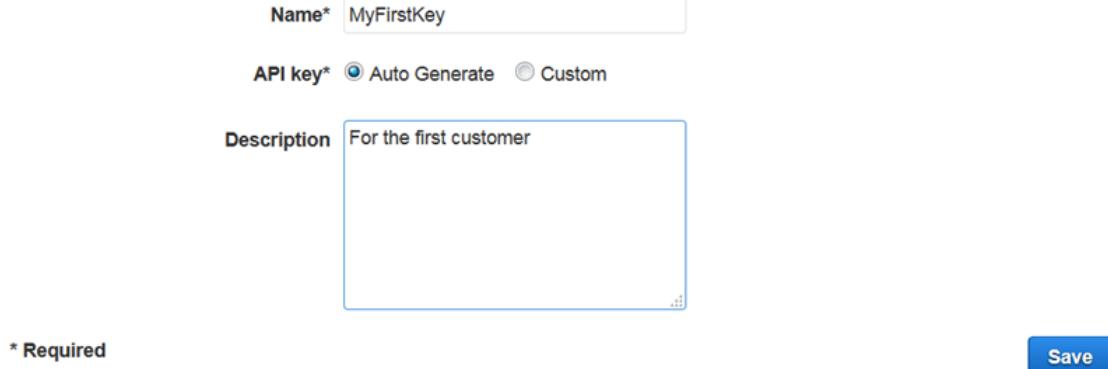
Create API Key

Name*

API key* Auto Generate Custom

Description

* Required Save



5. Repeat the preceding steps to create more API keys, if needed.

Import API Keys

The following procedure describes how to import API keys to use with usage plans.

To import API keys

1. In the main navigation pane, choose **API Keys**.
2. From the **Actions** drop-down menu, choose **Import API keys**.
3. To load a comma-separated key file, choose **Select CSV File**. You can also type the keys manually. For information about the file format, see [API Gateway API Key File Format \(p. 327\)](#).

Import API Keys

Use the field below to upload your existing API Keys as comma separated values (CSV). API Keys will be created in API Gateway and associated with a Usage Plan. Learn about the CSV format in the [API Gateway documentation](#).

[Select CSV File](#)

1	name,Key,Description(enabled,usageplanIds
	2 ImportedKey,CWaiyZjNC212f9P7hcxG17Ae803jEdFu8pzfryqf,an imported key,true,abcdef

Fail on warnings Ignore warnings [Import](#)

4. 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 error.
5. To start importing the selected API keys, choose **Import**.

Now that you've set up the API key, you can proceed to [create and use a usage plan \(p. 320\)](#).

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 \(p. 317\)](#).

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 \(If Needed\) \(p. 320\)](#)
- [Create Usage Plans \(p. 321\)](#)
- [Test a Usage Plan \(p. 323\)](#)
- [Maintain Plan Usage \(p. 323\)](#)

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 may 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 will include 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 Usage Plans

The following procedure describes how to create a usage plan.

To create a usage plan

1. In the Amazon API Gateway main navigation pane, choose **Usage Plans**, and then choose **Create**.
2. Under **Create Usage Plan**, do the following:
 - a. For **Name**, type a name for your plan (for example, `Plan_A`).
 - b. For **Description**, type a description for your plan.
 - c. Select **Enable throttling**, and set **Rate** (for example, `100`) and **Burst** (for example, `200`).
 - d. Choose **Enable quota**, and set its limit (for example, `5000`) for a selected time interval (for example, `Month`).
 - e. Choose **Save**.

Amazon API Gateway Developer Guide
Create, Configure, and Test Usage
Plans with the API Gateway Console

The screenshot shows the 'Create Usage Plan' page. On the left, a sidebar lists 'APIs' (PetShop, PetStore), 'Usage Plans' (selected), 'API Keys', 'Custom Domain Names', 'Client Certificates', and 'Settings'. The main area has tabs for 'Usage Plans' (selected) and 'Create'. A 'Create' button is at the top right. The central panel is titled 'Create Usage Plan' and contains the following fields:

- Name***: Plan_A
- Description**: First usage plan
- Throttling**:
 - Enable throttling: checked
 - Rate: 100 requests per second
 - Burst: 200 requests
- Quota**:
 - Enable quota: checked
 - 5000 requests per Month

At the bottom are 'Required' and 'Save' buttons.

3. To add a stage to the plan, do the following in the **Associated API Stages** pane:
 - a. Choose **Add API Stage**.
 - b. Choose an API (for example, **PetStore**) from the **API** drop-down list.
 - c. Choose a stage (for example, **Stage_1**) from the **Stage** drop-down list.
 - d. Choose the checkmark icon to save.
 - e. Choose **Next**.

Associated API Stages

Associate API stages to this usage plan. Subscribers will only be allowed to access the API stages that are associated with the plan. Choose "Add Stage" below, then use the dropdown to select an API and stage to enable for this usage plan.

The dialog has a header 'Add API Stage' and a table with columns 'API' and 'Stage'. It shows 'PetStore' selected in the API dropdown and 'testStage' selected in the Stage dropdown. There are 'Back' and 'Next' buttons at the bottom.

API	Stage
PetStore	testStage

Back Next

4. To add a key to the plan, do the following in the **Usage Plan API Keys** pane:

- a. To use an existing key, choose **Add API Key to Usage Plan**.
- b. For **Name**, type a name for the key you want to add (for example, **MyFirstKey**).
- c. Choose the checkmark icon to save.
- d. As needed, repeat the preceding steps to add other existing API keys to this usage plan.

Usage Plan API Keys

Subscribe an API key to this usage plan. Choose "Add API Key" below to search through your existing API keys. Once a key is associated with a plan, API Gateway will meter all requests from the key and apply the plan's throttling and quota limits.

The screenshot shows a user interface for managing API keys in a usage plan. At the top, there are two buttons: 'Add API Key to Usage Plan' and 'Create API Key and add to Usage Plan'. Below these is a dropdown menu set to 'Results per page 100'. A table header row contains the word 'Name'. In the main body, there is a single entry: 'MyFirstKey (Hiorr...)' with a checkbox next to it, which is checked. At the bottom of the table are navigation arrows ('<', '>') and page number ('Page 1'). Below the table are two buttons: 'Back' and 'Done'.

Note

To add a new API key to the usage plan, choose **Create API Key and add to Usage Plan** and follow the instructions.

5. To finish creating the usage plan, choose **Done**.
6. If you want to add more API stages to the usage plan, choose **Add API Stage** to repeat the preceding steps.

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 \(p. 326\)](#).

Maintain Plan Usage

Maintaining a usage plan involves monitoring the used and remaining quotas over a given time period and extending the remaining quotas by a specified amount. The following procedures describe how to monitor and extend quotas.

To monitor used and remaining quotas

1. In the API Gateway main navigation pane, choose **Usage Plans**.
2. Choose a usage plan from the list of the usage plans in the secondary navigation pane in the middle.
3. From within the specified plan, choose **API Keys**.
4. Choose an API key, and then choose **Usage** to view **Subscriber's Traffic** from the plan you're monitoring.

5. Optionally, choose **Export**, choose a **From** date and a **To** date, 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.

To extend the remaining quotas

1. Repeat steps 1–3 of the previous procedure.
2. On the usage plan page, choose **Extension** from the usage plan window.
3. Type a number for the **Remaining** request quotas.
4. Choose **Save**.

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.

For the REST API calls to create and deploy an API, see [restapi:create](#) and [deployment:create](#), respectively.

Topics

- [Require an API Key on a Method \(p. 325\)](#)
- [Create or Import API Keys \(p. 325\)](#)

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 \(p. 327\)](#).

With the API key created, you can now proceed to [Create, Configure, and Test Usage Plans Using the API Gateway REST API \(p. 325\)](#).

Create, Configure, and Test Usage Plans Using the API Gateway 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 \(p. 324\)](#).

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 \(p. 325\)](#)
- [Create a Usage Plan \(p. 325\)](#)
- [Manage a Usage Plan \(p. 326\)](#)
- [Test Usage Plans \(p. 326\)](#)

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 \(p. 320\)](#).

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=false HTTP/1.1
Host: apigateway.us-east-1.amazonaws.com
Content-Type: text/csv
Authorization: ...

key,name,description,enabled,usageplanIds
abcdef1234ghijklmnop8901234567, importedKey_1, firststone, 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`').

Manage a Usage Plan

The following procedure describes how to manage a usage plan.

To manage a usage plan with the REST API

1. Call [usageplan:by-id](#) to get a usage plan of a given plan ID. To see the available usage plans, call [apigateway:usage-plans](#).
2. Call [usageplan:update](#) to add a new API stage to the plan, replace an existing API stage in the plan, remove an API stage from the plan, or modify the rate limits or quotas.
3. Call [usage:get](#) to query the usage data in a specified time interval.
4. Call [usage:update](#) to grant an extension to the current usage in a usage plan.

Test Usage Plans

As an example, let's use the PetStore API, which was created in [Build an API Gateway API from an Example \(p. 9\)](#). 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, xbvxlpjch) 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: xbvxlpjch.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 your CloudWatch logs. You can ignore such occasional errors.

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
apikey1234abcdefg hij0123456789,MyFirstApiKey
```

A Key value must be between 30 and 128 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,apikey1234abcdefg hij0123456789,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,apikey1234abcdefg hij0123456789,"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.

Documenting an API Gateway API

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 Swagger 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 defined in an external Swagger file. Before explaining how to document your API, we'll show how API documentation is represented in API Gateway.

Topics

- [Representation of API Documentation in API Gateway \(p. 329\)](#)
- [Document an API Using the API Gateway Console \(p. 337\)](#)
- [Document an API Using the API Gateway REST API \(p. 345\)](#)
- [Publish API Documentation \(p. 360\)](#)
- [Import API Documentation \(p. 366\)](#)
- [Control Access to API Documentation \(p. 368\)](#)

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 \(p. 329\)](#)
- [Documentation Versions \(p. 336\)](#)

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 \(p. 331\)](#).

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 Swagger.",  
  "my-info" : "My custom info not recognized by Swagger."  
}
```

To set it as a value of `properties` using the API Gateway REST API, encode this object as a JSON string:

```
"{\n\t\"info\": {\n\t\t\"description\": \"My first API with Amazon API Gateway.\",\n\t}\n}"
```

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 Swagger and those that cannot. In the preceding example, `info`, `description`, and `x-custom-info` are recognized by Swagger as a standard Swagger object, property, or extension. In contrast, `my-info` is not compliant with the

Swagger specification. API Gateway propagates Swagger-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 Swagger 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 API entities of the API, AUTHORIZER, METHOD, MODEL, REQUEST_BODY or RESOURCE type.

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	Remark
1	/pets	*	id	Documentation associated with this location pattern will be inherited by entities matching the location pattern.
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 1.
3	/pets/ petId	*	id	Documentation associated with this location

Case	path	statusCode	name	Remark
				<p>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 2 and is more specific than Case 1.</p>

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 Swagger 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.
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 ", ... } }	type	The default values of path, and method are /and *, respectively.	Yes, matching path by prefix, and matching

API entity	Valid location fields	Required location fields	Default field values	Inheritable Content
	<pre> "path": "resource_path", "method": "http_verb" }, ... } </pre>			method by exact values.
Request header parameter	<pre> { "location": { "type": "REQUEST_HEADER", "path": "resource_path", "method": "HTTP_verb", "name": "header_name" }, ... } </pre>	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	<pre> { "location": { "type": "PATH_PARAMETER", "path": "resource/ path_parameter_name", "method": "HTTP_verb", "name": "path_parameter_name" }, ... } </pre>	type, name	The default values of path and method are / and *, respectively.	Yes, matching path by prefix and matching method by exact values.
Response	<pre> { "location": { "type": "RESPONSE", "path": "resource_path", "method": "http_verb", "statusCode": "status_code" }, ... } </pre>	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.

API entity	Valid location fields	Required location fields	Default field values	Inheritable Content
Response header	<pre>{ "location": { "type": "RESPONSE_HEADER", "path": "resource_path", "method": "http_verb", "statusCode": "status_code", "name": "header_name" }, ... }</pre>	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.
Response body	<pre>{ "location": { "type": "RESPONSE_BODY", "path": "resource_path", "method": "http_verb", "statusCode": "status_code" }, ... }</pre>	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.
Authorizer	<pre>{ "location": { "type": "AUTHORIZER", "name": "authorizer_name" }, ... }</pre>	type	N/A	No
Model	<pre>{ "location": { "type": "MODEL", "name": "model_name" }, ... }</pre>	type	N/A	No

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 Swagger 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.

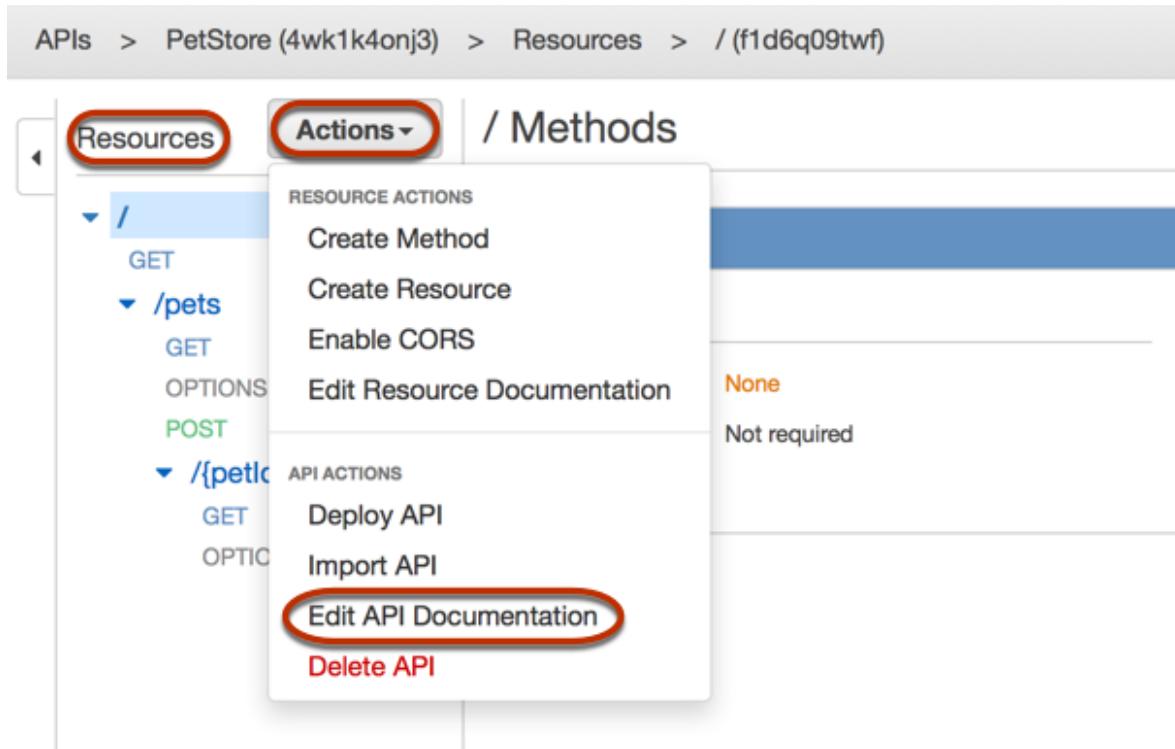
A prerequisite for creating and editing the documentation of an API is that you must have already created the API. In this section, we use the [PetStore](#) API as an example. To create an API using the API Gateway console, follow the instructions in [Build an API Gateway API from an Example \(p. 9\)](#).

Topics

- [Document the API Entity \(p. 337\)](#)
- [Document a RESOURCE Entity \(p. 340\)](#)
- [Document a METHOD Entity \(p. 340\)](#)
- [Document a QUERY_PARAMETER Entity \(p. 341\)](#)
- [Document a PATH_PARAMETER Entity \(p. 342\)](#)
- [Document a REQUEST_HEADER Entity \(p. 342\)](#)
- [Document a REQUEST_BODY Entity \(p. 343\)](#)
- [Document a RESPONSE Entity \(p. 343\)](#)
- [Document a RESPONSE_HEADER Entity \(p. 343\)](#)
- [Document a RESPONSE_BODY Entity \(p. 344\)](#)
- [Document a MODEL Entity \(p. 344\)](#)
- [Document an AUTHORIZER Entity \(p. 345\)](#)

Document the API Entity

To add a documentation part for the API entity, choose **Resources** from the **PetStore** API. Choose the **Actions** → **Edit API Documentation** menu item.



If a documentation part was not created for the API, you get the documentation part's properties map editor. Type the following properties map in the text editor and then choose **Save** to create the documentation part.

```
{  
  "info": {  
    "description": "Your first API Gateway API.",  
    "contact": {  
      "name": "John Doe",  
      "email": "john.doe@api.com"  
    }  
  }  
}
```

Note

You do not encode the properties map into a JSON string, as you must do when using the API Gateway REST API. The API Gateway console stringifies the JSON object for you.

Documentation

Provide your API documentation in JSON format in the form below.

Type API

```
1  {
2      "info": {
3          "description" : "Your first API Gateway API.",
4          "contact": {
5              "name": "John Doe",
6              "email": "john.doe@api.com"
7          }
8      }
9 }
```

Close **Save**

If a documentation part has already been created, you first get the properties map viewer, as shown in the following.

Documentation

Specify your documentation part in JSON format in the form below. For more information, see the [Documentation Parts Documentation](#).

Type API

```
{  
  "info": {  
    "description": "Your first API Gateway API.",  
    "contact": {  
      "name": "John Doe",  
      "email": "john.doe@api.com"  
    }  
  }  
}
```

Delete **Close** **Edit**

Choosing **Edit** brings up the properties map editor as shown previously.

Document a RESOURCE Entity

To add or edit the documentation part for the API's root resource, choose **/** under the **Resource** tree, and then choose the **Actions** → **Edit Resource Documentation** menu item.

If no documentation part was created for this entity, you get the **Documentation** window. Type a valid properties map in the editor. Then choose **Save** and **Close**.

```
{  
  "description": "The PetStore's root resource."  
}
```

If a documentation part has already been defined for the RESOURCE entity, you get the documentation viewer. Choose **Edit** to open the **Documentation** editor. Modify the existing properties map. Choose **Save** and then choose **Close**.

If necessary, repeat these steps to add a documentation part to other RESOURCE entities.

Document a METHOD Entity

To add or edit documentation for a METHOD entity, using the **GET** method on the root resource as an example, choose **GET** under the **/** resource and the choose the **Actions** → **Edit Method Documentation** menu item.

For the new documentation part, type the following properties map in the **Documentation** editor in the **Documentation** window. Then choose **Save** and **Close**.

```
{  
    "tags" : [ "pets" ],  
    "description" : "PetStore HTML web page containing API usage information"  
}
```

For the existing documentation, choose **Edit** from the **Documentation** viewer. Edit the documentation content in the **Documentation** editor and choose **Save**. Choose **Close**.

From the **Documentation** viewer, you can also delete the documentation part.

If necessary, repeat these steps to add a documentation part to other methods.

Document a QUERY_PARAMETER Entity

To add or edit a documentation part for a request query parameter, using the `GET /pets?type=...&page=...` method as an example, choose **GET** under `/pets` from the **Resources** tree. Choose **Method Request** in the **Method Execution** window. Expand the **URL Query String Parameters** section. Choose the `page` query parameter, for example, and choose the book icon to open the **Documentation** viewer or editor.

The screenshot shows the AWS Lambda Function Configuration interface for a function named 'HelloWorld'. At the top, there are tabs for 'Overview', 'Code', 'Logs', and 'Environment'. Below these are sections for 'Handler' (set to 'index.handler') and 'Runtime' (set to 'Node.js 12.x'). Under the 'Configuration' tab, there is a 'Function URL' section with a 'Create' button. The main area contains several configuration groups: 'Memory Size', 'Timeout', 'Tracing', 'Environment Variables', and 'Environment'. In the 'Environment' group, there is a 'Variables' table:

Name	Type	Value
stage	String	prod
awscloudfrontoriginid	String	12345678901234567890123456789012
awscloudfrontoriginport	String	443
awscloudfrontoriginregion	String	us-east-1
awscloudfrontoriginscheme	String	https
awscloudfrontoriginidalias	String	prod
awscloudfrontoriginportalias	String	443
awscloudfrontoriginregionalias	String	us-east-1
awscloudfrontoriginschemalias	String	https

At the bottom of the configuration pane, there is a 'Save' button.

Alternatively, you can choose **Documentation** under the **PetStore** API from the main navigation pane. Then choose **Query Parameter** for **Type**. For the PetStore example API, this shows the documentation parts for the `page` and `type` query parameters.

Documentation

[Create Documentation Part](#)
[Import Documentation](#)
[Publish Documentation](#)

Add documentation to help developers understand how to interact with your API. Documentation parts can be shared across multiple resources and methods by specifying a wildcard value (*) for method or status code, eg. documentation for a 200 response can be used in multiple locations. You can also import documentation by supplying a Swagger definition file, and publish documentation to a stage. For more information, reference the [documentation](#).

The screenshot shows the 'Documentation' section of the Amazon API Gateway Developer Guide. At the top, there are buttons for 'Create Documentation Part', 'Import Documentation', and 'Publish Documentation'. Below this, a text area explains how to add documentation to help developers understand API interactions. A search bar at the top right includes fields for 'Type' (set to 'Query Parameter'), 'Path', 'Method' (set to 'All'), and 'Name'. Two documentation parts are listed in a grid:

- Documentation Part 1:** Type: Query Parameter, Path: /pets, Method: GET, Name: page. Description: "Page number of results to return." (with a JSON schema: { "description": "Page number of results to return." })
- Documentation Part 2:** Type: Query Parameter, Path: /pets, Method: GET, Name: type. Description: "The type of pet to retrieve" (with a JSON schema: { "description": "The type of pet to retrieve" })

Each documentation part has 'Edit' and 'Clone' buttons at the bottom.

For an API with query parameters defined for other methods, you can filter your selection by specifying the path of the affected resource for **Path**, choosing the desired HTTP method from **Method**, or typing the query parameter name in **Name**.

For example, choose the page query parameter. Choose **Edit** to modify the existing documentation. Choose **Save** to save the change.

To add a new documentation part for a QUERY_PARAMETER entity, choose **Create Documentation Part**. Choose **Query Parameter** for **Type**. Type a resource path (e.g., /pets) in **Path**. Choose an HTTP verb (e.g., **GET**) for **Method**. Type a properties description in the text editor. Then choose **Save**.

If necessary, repeat these steps to add a documentation part to other request query parameters.

Document a PATH_PARAMETER Entity

To add or edit documentation for a path parameter, go to **Method Request** of the method on the resource specified by the path parameter. Expand the **Request Paths** section. Choose the book icon for the path parameter to open the **Documentation** viewer or editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the **PetStore** API from the main navigation pane. Choose **Path Parameter** for **Type**. Choose **Edit** on a path parameter from the list. Modify the content and then choose **Save**.

To add documentation for a path parameter not listed, choose **Create Documentation Part**. Choose **Path Parameter** for **Type**. Set a resource path in **Path**, choose a method from **Method**, and set a path parameter name in **Name**. Add the documentation's properties and choose **Save**.

If required, repeat these steps to add or edit a documentation part to other path parameters.

Document a REQUEST_HEADER Entity

To add or edit documentation for a request header, go to **Method Request** of the method with the header parameter. Expand the **HTTP Request Headers** section. Choose the book icon for the header to open the **Documentation** viewer or editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Request Header** for **Type**. Choose **Edit** on a listed request header to change the documentation. To

add documentation for an unlisted request header, choose **Create Documentation Part**. Choose **Request Header** for **Type**. Specify a resource path in **Path**. Choose a method for **Method**. Type a header name in **Name**. Then add and save a properties map.

If required, repeat these steps to add or edit a documentation part to other request headers.

Document a REQUEST_BODY Entity

To add or edit documentation for a request body, go to **Method Request** for a method. Choose the book icon for **Request Body** to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Request Body** for **Type**. Choose **Edit** on a listed request body to change the documentation. To add documentation for an unlisted request body, choose **Create Documentation Part**. Choose **Request Body** for **Type**. Set a resource path in **Path**. Choose an HTTP verb for **Method**. Then add and save a properties map.

If required, repeat these steps to add or edit a documentation part to other request bodies.

Document a RESPONSE Entity

To add or edit documentation for a response, go to **Method Response** of a method. Choose the book icon for **Method Response** to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

The screenshot shows the AWS API Gateway interface for documenting a method response. At the top, there's a navigation bar with 'Method Execution' and 'Method Response' tabs, where 'Method Response' is active and highlighted with a red oval. Below the tabs, there's a section for 'Provide information about this method's response types, their headers and content types.' Under this, there's a table with a single row for 'HTTP Status'. The table has columns for 'Status Code' (containing '200') and 'Actions' (with icons for edit and delete). At the bottom left, there's a blue 'Add Response' button.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Response (status code)** for **Type**. Choose **Edit** on a listed response of a specified HTTP status code to change the documentation. To add documentation for an unlisted response body, choose **Create Documentation Part**. Choose **Response (status code)** for **Type**. Set a resource path in **Path**. Choose an HTTP verb for **Method**. Type an HTTP status code in **Status Code**. Then add and save the documentation part properties.

If required, repeat these steps to add or edit a documentation part to other responses.

Document a RESPONSE_HEADER Entity

To add or edit documentation for a response header, go to **Method Response** of a method. Expand a response section of a given HTTP status. Choose the book icon for a response header under **Response Headers for StatusCode** to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Response Header** for **Type**. Choose **Edit** on a listed response header to change the documentation. To add documentation for an unlisted response header, choose **Create Documentation Part**. Choose

Response Header for Type. Set a resource path in **Path**. Choose an HTTP verb for **Method**. Type an HTTP status code in **Status Code**. Type the response header name in **Name**. Then add and save the documentation part properties.

If required, repeat these steps to add or edit a documentation part to other response headers.

Document a RESPONSE_BODY Entity

To add or edit documentation for a response body, go to **Method Response** of a method. Expand the response section of a given HTTP status. Choose the book icon for **Response Body for Status Code** to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Response Body for Type**. Choose **Edit** on a listed response body to change the documentation. To add documentation for an unlisted response body, choose **Create Documentation Part**. Choose **Response Body for Type**. Set a resource path in **Path**. Choose an HTTP verb for **Method**. Type an HTTP status code in **Status Code**. Then add and save the documentation part properties.

If required, repeat these steps to add or edit a documentation part to other response bodies.

Document a MODEL Entity

Documenting a **MODEL** entity involves creating and managing **DocumentationPart** 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 or edit documentation for a model, go to **Models** of the API in the main navigation pane. Choose the book icon for the name of a listed model to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Model for Type**. Choose **Edit** on a listed model to change the documentation. To add documentation for an unlisted model, choose **Create Documentation Part**. Choose **Model for Type**. Give a name to the model in **Name**. Then add and save the documentation part properties.

If required, repeat these steps to add or edit a documentation part to other models.

Document an AUTHORIZER Entity

To add or edit documentation for an authorizer, go to **Authorizers** for the API in the main navigation pane. Choose the book icon for the listed authorizer to open the **Documentation** viewer and then editor. Add or modify the properties of the documentation part.

Alternatively, choose **Documentation** under the API from the main navigation pane. Then choose **Authorizer for Type**. Choose **Edit** on a listed authorizer to change the documentation. To add documentation for an unlisted authorizer, choose **Create Documentation Part**. Choose **Authorizer for Type**. Give a name to the authorizer in **Name**. Then add and save the documentation part properties.

If required, repeat these steps to add or edit a documentation part to other authorizers.

To add a documentation part for an authorizer, choose **Create Documentation Part**. Choose **Authorizer for Type**. Specify a value for the valid location field of **Name** for the authorizer.

Add and save the documentation content in the properties map editor.

If required, repeat these steps to add a documentation part to another authorizer.

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 [Build an API Gateway API from an Example \(p. 9\)](#).

Topics

- [Document the API Entity \(p. 346\)](#)
- [Document a RESOURCE Entity \(p. 347\)](#)
- [Document a METHOD Entity \(p. 349\)](#)
- [Document a QUERY_PARAMETER Entity \(p. 352\)](#)
- [Document a PATH_PARAMETER Entity \(p. 353\)](#)
- [Document a REQUEST_BODY Entity \(p. 353\)](#)
- [Document a REQUEST_HEADER Entity \(p. 354\)](#)

- [Document a RESPONSE Entity \(p. 355\)](#)
- [Document a RESPONSE_HEADER Entity \(p. 356\)](#)
- [Document an AUTHORIZER Entity \(p. 357\)](#)
- [Document a MODEL Entity \(p. 358\)](#)
- [Update Documentation Parts \(p. 360\)](#)
- [List Documentation Parts \(p. 360\)](#)

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: YYYYMMDDTtttttz
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\nGateway.\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\nGateway.\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: YYYYMMDDTtttttz
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\nGateway.\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: YYYYMMDDTTttttZ
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\"\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\"\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\nparameter.\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\nparameter.\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: YYYYMMDDTtttttz
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": {
        "curries": {
            "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
            "name": "documentationpart",
            "templated": true
        },
        "self": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        }
    },
    "id": "o64bjj",
    "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": {
        "curries": {
            "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
            "name": "documentationpart",
            "templated": true
        },
        "self": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        }
    },
    "id": "o64bjj",
    "location": {
        "path": "/pets",
        "method": "GET",
        "name": null,
        "statusCode": null,
        "type": "METHOD"
    },
    "properties": "{\n\t\"summary\" : \"List all pets.\n}"
}
```

```

    "id": "o64bjj",
    "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](#) 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: YYYYMMDDTtttttZ
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/o64bjj"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/o64bjj"
        }
    },
    "id": "o64bjj",
    "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: YYYYMMDDTtttttz
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/partsh9ht5w"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/partsh9ht5w"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/partsh9ht5w"
        }
    },
    "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: YYYYMMDDTtttttZ
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\t\"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\ninvocation.\n\"}\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\ninvocation.\n\"}\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: YYYYMMDDTtttttz
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: YYYYMMDDTtttttZ
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\"\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 [DocumentationPart](#) 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/partslkn4uq"
        },
        "documentationpart:delete": {
            "href": "/restapis/4wk1k4onj3/documentation/partslkn4uq"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/partslkn4uq"
        }
    },
    "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

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 \(p. 361\)](#)

- [Create a Documentation Snapshot \(p. 361\)](#)
- [Update a Documentation Snapshot \(p. 362\)](#)
- [Get a Documentation Snapshot \(p. 362\)](#)
- [Associate a Documentation Snapshot with an API Stage \(p. 362\)](#)
- [Download a Documentation Snapshot Associated with a Stage \(p. 363\)](#)

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: YYYYMMDDTtttttZ
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: YYYYMMDDTtttttZ
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: <YYYYMMDDTtttttz>
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: <YYYYMMDDTtttttz>
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: <YYYYMMDDTtttttz>
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"  
        }  
    ]  
}
```

The following procedure describes how to publish a documentation version.

To publish a documentation version using the API Gateway console

1. Choose **Documentation** for the API from the main navigation pane in the API Gateway console.
2. Choose **Publish Documentation** in the **Documentation** pane.
3. Set up the publication:
 - a. Choose an available name for **Stage**.
 - b. Type a version identifier, e.g., 1.0.0, in **Version**.
 - c. Optionally, provide a description about the publication in **Description**.
4. Choose **Publish**.

You can now proceed to download the published documentation by exporting the documentation to an external Swagger file.

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 Swagger 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: YYYYMMDDTtttttZ  
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 Swagger. 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 a Swagger 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 Swagger file.

```
{
  "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 a Swagger-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 Swagger 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 Swagger specification and is not injected into its associated API entity definitions.

If a documentation-rendering tool (e.g., [Swagger UI](#)) parses the API entity definitions to extract documentation attributes, any non Swagger-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 `documentationpart: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 Swagger 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: YYYYMMDDTtttttZ
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 Swagger 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 API Using the API Gateway Console \(p. 415\)](#).

Import API Documentation

As with importing API entity definitions, you can import documentation parts from an external Swagger file into an API in API Gateway. You specify the to-be-imported documentation parts within the [x-amazon-apigateway-documentation Object \(p. 492\)](#) extension in a valid Swagger 2.0 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 Swagger 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 Swagger 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.

```
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"  
    ]  
}
```

```
"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 Swagger 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. Choose **Documentation** for the API from the main navigation pane on the console.
2. Choose **Import Documentation** in the **Documentation** pane.
3. Choose **Select Swagger File** to load a file from a drive, or copy and paste a 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 \(p. 366\)](#).
4. Optionally, choose **Fail on warnings** or **Ignore warnings**, and choose **Merge** or **Overwrite** from **Import mode**.
5. 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 [Control Who Can Create and Manage an API Gateway API with IAM Policies \(p. 254\)](#).

Deploying an API in Amazon API Gateway

After creating your API, you must deploy it to make the API callable for your users.

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 the client to call. Every time you update an API, which includes modification of methods, integrations, authorizers, and anything else other than stage settings, you must redeploy the API to an existing stage or to a new stage. 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 \(p. 397\)](#), enabling 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 method URL. The method URL is determined by an API's host name, a stage name, and a resource path. The host name and the stage name determine the API's base URL.

Using the API's default domain name, the base URL of an API in a given stage (`{stageName}`) is of the following format:

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stageName}
```

To make an API's default base 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. 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 becomes the following:

```
https://{api.example.com}/{basePath}
```

For each stage, you can optimize the API performance by adjusting the default account-level request throttling limits and enabling API caching. You can also enable logging API calls to CloudTrail or CloudWatch and 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 run time. At an API stage, you can export the API definitions and generate an SDK for your users to call the API using a supported programming language.

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 \(p. 372\)](#) value from the stage name of `test` to that of `prod`.

You can also include a [canary release](#) for testing new changes. This is referred to as a canary release deployment. It makes available a base version and updated versions of the API on the same stage, allowing you to introduce new features in the same environment for the base version. For more information, see [the section called "Set up a Canary Release Deployment" \(p. 397\)](#).

In this section, we discuss how to deploy an API, using the [API Gateway console](#) or calling the [API Gateway REST API](#). To use other tools to do the same, see the documentation of, for example, [AWS CLI](#) or an [AWS SDK](#).

To monetize your API deployment, you can leverage the API Gateway integration with AWS Marketplace to vend your API as a Software as a Service (SaaS) product. The instructions are also included in this chapter.

Topics

- [Create a Deployment in API Gateway \(p. 371\)](#)
- [Set up a Stage in API Gateway \(p. 373\)](#)
- [Set up an API Gateway Canary Release Deployment \(p. 397\)](#)
- [Export an API from API Gateway \(p. 414\)](#)
- [Generate SDK of an API \(p. 416\)](#)
- [Set up Custom Domain Name for an API in API Gateway \(p. 432\)](#)
- [Sell Your API Gateway API through AWS Marketplace \(p. 453\)](#)

Create a Deployment in API Gateway

In API Gateway, a deployment is represented by a [Deployment](#) resource. It is like an executable of an API represented by a [RestApi](#) resource. For the client to call your API, you must create a deployment and associate a stage to it. A stage is represented by a [Stage](#) resource and represents a snapshot of the API, including methods, integrations, models, mapping templates, Lambda authorizers (formerly known as custom authorizers), etc. 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](#)” (p. 373).

Topics

- [Create a Deployment Using AWS CLI \(p. 371\)](#)
- [Deploy API from the API Gateway Console \(p. 372\)](#)

Create a Deployment Using AWS CLI

Creating an deployment amounts to instantiating the [Deployment](#) resource. You can use the API Gateway console, AWS CLI, an AWS SDK or the API Gateway REST API to create an deployment.

To use 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 so 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> \  
  --patch-operations op='replace',path='/deploymentId',value='<deployment-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.

Deploy API from the API Gateway Console

You must have created an API before deploying it for the first time. For more information see [Creating an API in Amazon API Gateway \(p. 81\)](#).

Topics

- [Deploy an API to a Stage \(p. 372\)](#)
- [Update the Stage Configuration of a Deployment \(p. 372\)](#)
- [Set Stage Variables for the Deployment \(p. 372\)](#)
- [Associate a Stage with a Different Deployment \(p. 373\)](#)

Deploy an 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 Deployment \(p. 373\)](#) 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** navigation pane, choose **Actions**.
4. From the **Actions** drop-down menu, choose **Deploy API**.
5. In the **Deploy API** dialog, choose an entry from the **Deployment stage** dropdown list.
6. If you choose **[New Stage]**, type a name in **Stage name** and optionally provide a description for the stage and deployment in **Stage description** and **Deployment description**. If you choose an existing stage, you may want to provide a description of the new deployment in **Deployment description**.
7. Choose **Deploy** to deploy the API to the specified stage with default stage settings.

Update the Stage Configuration of a Deployment

After an API is deployed, you can modify the stage settings to enable or disable 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 run time. For more information, see [Update Stage Settings \(p. 374\)](#).

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 will not be logged even if you have enabled the logging option.

Set Stage Variables for the Deployment

For a deployment, you can set or modify stage variables to pass deployment-specific data to the API integration at run time. You can do this on the **Stage Variables** tab in the **Stage Editor**. For more information, see instructions in [Set up Stage Variable for API Deployment \(p. 385\)](#).

Associate a Stage with a Different 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 not already in **Stage Editor**, choose the stage you want to update the deployment from an API's **Stages** option in the APIs main navigation pane.
2. On the **Deployment History** tab, choose the option button next to the deployment you want the stage to use.
3. Choose **Change Deployment**.

Set up a Stage in API Gateway

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 set up stage settings to enable caching, customize request throttling, configure logging, define stage variables or attach a canary release for testing.

Topics

- [Set up a Stage Using the API Gateway Console \(p. 373\)](#)
- [Throttle API Requests for Better Throughput \(p. 376\)](#)
- [Enable API Caching to Enhance Responsiveness \(p. 378\)](#)
- [Set up API Logging in API Gateway \(p. 382\)](#)
- [Set up Stage Variable for API Deployment \(p. 385\)](#)
- [Set up Tags for an API Stage in API Gateway \(p. 395\)](#)

Set up a Stage Using the API Gateway Console

Topics

- [Create a New Stage \(p. 373\)](#)
- [Update Stage Settings \(p. 374\)](#)
- [Delete a Stage for an API \(p. 376\)](#)

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 and use a new stage or 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 do so using the API Gateway console, follow the instructions below.

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. From the APIs navigation pane, choose **Stages** under an API.
3. From the **Stages** navigation pane, choose **Create**.

4. Under **Create Stage**, type a stage name, e.g., prod, for **Stage name**.
5. Optionally, type a stage description for **Stage description**
6. From the **Deployment** drop-down list, choose the date and time of the existing API deployment you want to associate with this stage.
7. Choose **Create**.

Update Stage Settings

After a successful deployment of an API, the stage is populated with default settings. You can use the console or API Gateway REST API to change the stage settings, including API caching and logging. In the following, we show how to do so using the **Stage Editor** of the API Gateway console.

Update Stage Settings Using the API Gateway Console

Make sure that you have already deployed the API and created a stage at least once before proceeding.

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the **APIs** pane, choose the API you want to update the stage settings, and then choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. In the **Stage Editor** pane, choose the **Settings** tab.
5. To enable API caching, select the **Enable API cache** option under the **Cache Settings** section. Then, choose desired options and associated values for **Cache capacity**, **Encrypt cache data**, **Cache time-to-live (TTL)**, as well as the requirements for per-key cache invalidation. For more information about the stage-level cache settings, see [Enable API Caching \(p. 378\)](#).

Important

By selecting this option, your AWS account may be charged for API caching.

Tip

You can override enabled stage-level cache settings. To do so, expand the stage under the **Stages** secondary navigation pane and choose a method. Then, in the stage editor, choose the **Override for this method** option for **Settings**. In the ensuing **Cache Settings** area, clear **Enable Method Cache** or customize any other desired options, before choosing **Save Changes**. For more information about the method-level cache settings, see [Enable API Caching \(p. 378\)](#).

6. To enable Amazon CloudWatch Logs for all of the methods associated with this stage of this API Gateway API, do the following:
 - a. Under the **CloudWatch Settings** section, select the **Enable CloudWatch Logs** option.

Tip

To enable method-level CloudWatch settings, expand the stage under the **Stages** secondary navigation pane, choose each method of interest, and, back in the stage editor, choose **Override for this method** for **Settings**. In the ensuing **CloudWatch Settings** area, make sure to select **Log to CloudWatch Logs** and any other desired options, before choosing **Save Changes**.

Important

Your account will be charged for accessing method-level CloudWatch metrics, but not the API- or stage- level metrics.

- b. For **Log level**, choose **ERROR** to write only error-level entries to CloudWatch Logs, or choose **INFO** to include all **ERROR** events as well as extra informational events.
- c. To log full API call request and response information, select **Log full requests/responses data**. No sensitive data will be logged unless the **Log full requests/responses data** option is selected.
- d. To have API Gateway to report to CloudWatch the API metrics of **API calls**, **Latency**, **Integration latency**, **400 errors** and **500 errors**, select the **Enable Detailed**

CloudWatch Metrics option. For more information about CloudWatch, see the [Amazon CloudWatch User Guide](#).

- e. Choose **Save Changes**. The new settings will take effect after a new deployment.

Important

To enable CloudWatch Logs for all or only some of the methods, you must also specify the ARN of an IAM role that enables API Gateway to write information to CloudWatch Logs on behalf of your IAM user. To do so, choose **Settings** from the APIs main navigation pane. Then type the ARN of an IAM role in the **CloudWatch log role ARN** text field. 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](#).

7. To set the stage-level throttle limit for all of the methods associated with this API, do the following in the **Default Method Throttling** section:
 - a. For **Rate**, type the maximum number of stage-level steady-state requests per second that API Gateway can serve without returning a 429 Too Many Requests response. This stage-level rate limit must not be more than the [account-level \(p. 376\)](#) rate limit as specified in [API Gateway Limits for Configuring and Running an API \(p. 582\)](#).
 - b. For **Burst**, type the maximum number of stage-level concurrent requests that API Gateway can serve without returning a 429 Too Many Requests response. This stage-level burst must not be more than the [account-level \(p. 376\)](#) burst limit as specified in [API Gateway Limits for Configuring and Running an API \(p. 582\)](#).

8. To override the stage-level throttling for individual methods, expand the stage under the **Stages** secondary navigation pane, choose a method of interest, and, back in the stage editor, choose **Override for this method** for **Settings**. In the **Default Method Throttling** area, select appropriate options.

Delete a Stage for an API

When you no longer need a stage, you can delete it to avoid paying for unused resources. In the following, we explain how to use the API Gateway console to delete a stage .

Warning

Deleting a stage may 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 with the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the stage you want to delete, and then choose **Delete Stage**.
4. When prompted, choose **Delete**.

Throttle API Requests for Better Throughput

To prevent your API from being overwhelmed by too many requests, Amazon API Gateway throttles requests to your API using the [token bucket algorithm](#), where a token counts for a request. Specifically, API Gateway sets a limit on a steady-state rate and a burst of request submissions against all APIs in your account. In the token bucket algorithm, the burst is the maximum bucket size.

When request submissions exceed the steady-state request rate and burst limits, API Gateway fails the limit-exceeding requests and returns `429 Too Many Requests` error responses to the client. Upon catching such exceptions, the client can resubmit the failed requests in a rate-limiting fashion, while complying with the API Gateway throttling limits.

As an API developer, you can set the limits for individual API stages or methods to improve overall performance across all APIs in your account. Alternatively, you can enable [usage plans \(p. 314\)](#) to restrict client request submissions to within specified request rates and quotas. This restricts the overall request submissions so that they don't go significantly past the account-level throttling limits.

Account-Level Throttling

By default, API Gateway limits the steady-state request rate to 10,000 requests per second (rps). It limits the burst (that is, the maximum bucket size) to 5,000 requests across all APIs within an AWS account. In API Gateway, the burst limit corresponds to the maximum number of concurrent request submissions that API Gateway can fulfill at any moment without returning `429 Too Many Requests` error responses.

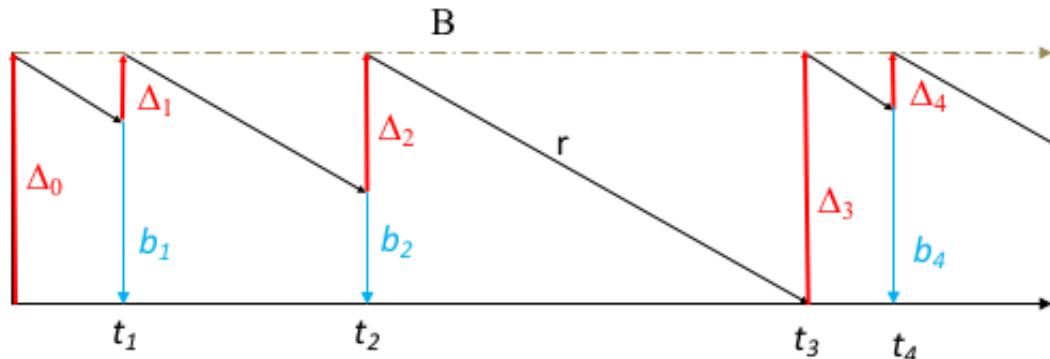
To help understand these throttling limits, here are a few examples, given the burst limit and the default account-level rate limit:

- If a caller submits 10,000 requests in a one second period evenly (for example, 10 requests every millisecond), API Gateway processes all requests without dropping any.
- If the caller sends 10,000 requests in the first millisecond, API Gateway serves 5,000 of those requests and throttles the rest in the one-second period.

- If the caller submits 5,000 requests in the first millisecond and then evenly spreads another 5,000 requests through the remaining 999 milliseconds (for example, about 5 requests every millisecond), API Gateway processes all 10,000 requests in the one-second period without returning 429 Too Many Requests error responses.
- If the caller submits 5,000 requests in the first millisecond and waits until the 101st millisecond to submit another 5,000 requests, API Gateway processes 6,000 requests and throttles the rest in the one-second period. This is because at the rate of 10,000 rps, API Gateway has served 1,000 requests after the first 100 milliseconds and thus emptied the bucket by the same amount. Of the next spike of 5,000 requests, 1,000 fill the bucket and are queued to be processed. The other 4,000 exceed the bucket capacity and are discarded.
- If the caller submits 5,000 requests in the first millisecond, submits 1,000 requests at the 101st millisecond, and then evenly spreads another 4,000 requests through the remaining 899 milliseconds, API Gateway processes all 10,000 requests in the one-second period without throttling.

More generally, at any given moment, when a bucket contains b and the maximum bucket capacity is B , the maximum additional tokens that can be added to the bucket is $\# = B - b$. This maximum number of additional tokens corresponds to the maximum number of additional concurrent requests that a client can submit without receiving any 429 error responses. In general, $\#$ varies in time. The value ranges from zero when the bucket is full (that is, $b=B$) to B when the bucket is empty (that is, $b=0$). The range depends on the request-processing rate, which is the rate at which tokens are removed from the bucket, and the rate limit rate, which is the rate at which tokens are added to the bucket.

The following schematic shows the general behaviors of $\#$, the maximum additional concurrent requests, as a function of time. The schematic assumes that the tokens in the bucket decrease at a combined rate of r , starting from an empty bucket.



The account-level rate limit can be increased upon request. To request an increase of account-level throttling limits, contact the [AWS Support Center](#). For more information, see [API Gateway Limits \(p. 582\)](#).

Note

The burst limit cannot be increased.

Default Method Throttling

As an API owner, you can set the default method throttling to override the account-level request throttling limits for a specific stage or for individual methods in an API. The Default method throttling limits are bounded by the account-level rate limits, even if you set the default method throttling limits higher than the account-level limits.

You can set the default method throttling limits by using the API Gateway console or by calling the [API Gateway REST API \(p. 581\)](#). For instructions on using the console, see [Update Stage Settings \(p. 374\)](#).

Enable API Caching to Enhance Responsiveness

You can enable API caching in Amazon API Gateway to cache your endpoint's response. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of the 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.

Important

Only `GET` methods should be cached.

Note

Caching is charged by the hour and is not eligible for the AWS free tier.

Enable Amazon API Gateway Caching

In API Gateway, you can enable caching for all methods for a specified stage. When you enable caching, you must choose a cache capacity. In general, a larger capacity gives a better performance, but also costs more.

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 recreating a new one with a modified capacity. All existing cached data is deleted.

In the API Gateway console, you configure caching in the **Settings** tab of a named **Stage Editor**.

To configure API caching for a given stage:

1. Go to the API Gateway console.
2. Navigate to the **Stage Editor** for the stage for which you want to enable caching.
3. Choose **Settings**.
4. Select **Enable API cache**.
5. Wait for the cache creation to complete.

Note

Creating or deleting a cache takes about 4 minutes for API Gateway to complete. When cache is created, the **Cache status** value changes from `CREATE_IN_PROGRESS` to `AVAILABLE`. When cache deletion is completed, the **Cache status** value changes from `DELETE_IN_PROGRESS` to an empty string.

When you enable caching within a stage's **Cache Settings**, you enable caching for all methods in that stage.

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 Caching

If you want more granularity in your caching settings, you can override the stage-level caching for individual methods. This includes disabling caching for a specific method, increasing or decreasing its TTL period, and turning on or off encryption of the cached response. If you anticipate that a method 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. Choose **Stages** of an API from the main navigation pane.
2. Choose a method of the API in the chosen stage, from the secondary navigation pane.
3. Choose **Override for this method** in **Settings**.
4. Choose appropriate settings under the **Cache Settings** section (that is shown only if the stage-level caching is enabled).

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.

For example, suppose you have a request of 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` will be 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 will be cached and will be 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.

[Method Execution](#) /streams - GET - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Authorization Settings •

Authorization NONE

API Key Required false

▼ URL Query String Parameters •

Name	Caching	
query		

Add query string

► HTTP Request Headers

► Request Models [Create a Model](#) •

Flush the API Stage Cache in API Gateway

When API caching is enabled, you can flush your API stage's entire cache to ensure your API's clients get the most recent responses from your integration endpoints.

To flush the API stage cache, you can choose the **Flush entire cache** button under the **Cache Settings** section in the **Settings** tab in a stage editor of the API Gateway console. The cache-flushing operation is almost instantaneous. As a result, the cache status is **AVAILABLE** immediately after flushing.

Notice that flushing the cache will cause the responses to ensuing requests to be serviced from the backend until the cache is built up again. During this period, the number of requests sent to the integration endpoint may increase. That may affect the overall latency of your API.

Invalidate an API Gateway Cache Entry

A client of your API can invalidate an existing cache entry and reloads 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 user 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 caller, attach a policy of the following format to an IAM execution role for the user.

```
{  
    "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 cache for requests on the specified resource (or resources). To specify a group of targeted resources, use a wild card (*) 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 \(p. 252\)](#)

If you do not impose an InvalidateCache policy, any client can invalidate the API cache. If all or most of the clients invalidate the API cache, there could be significant latency impact on 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 **Handle unauthorized requests** in the API Gateway console.

test Stage Editor Delete Stage

● Invoke URL: <https://██████████b.execute-api.us-east-1.amazonaws.com/test>

Settings Stage Variables SDK Generation Export Deployment History

Configure the metering and caching settings for the **test** stage.

Cache Settings

Cache status AVAILABLE Flush entire cache

Enable API cache (highlighted)

Enabling API cache increases cost and is not covered by the free tier. [See pricing for more details](#)

Cache capacity 0.5GB ▼

Encrypt cache data

Cache time-to-live (TTL) 300 ▼

Per-key cache invalidation

Require authorization (highlighted)

Handle unauthorized requests Ignore cache control header; Add a warning in response header (highlighted)

CloudWatch Settings Ignore cache control header; Add a warning in response header Ignore cache control header Fail the request with 403 status code

Enable CloudWatch Logs ?

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`.

Set up API Logging in API Gateway

To help debug issues related to request execution or client access to your API, you can enable Amazon CloudWatch Logs to trace API calls. Once enabled, API Gateway will log API calls in CloudWatch. There are two types of API logging: 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 one, which could be managed by API Gateway. You can specify the access details by selecting `$context (p. 191)` variables, expressed in a format of your choosing, and by choosing a log group as the destination. To preserve uniqueness of each log, access log format must include `$context.requestId`.

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, provided that the `ARN` column is selected for display. 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 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
```

The continuation characters (\) are meant as a visual aid and the log format cannot have any line breaks.

- JSON:

```
{ "requestId": "$context.requestId", \
  "ip": "$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" \
}
```

The continuation characters (\) are meant as a visual aid and the log format cannot have any line breaks.

- XML:

```
<request id="$context.requestId"> \
  <ip>$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>
```

The continuation characters (\) are meant as a visual aid and the log format cannot have any line breaks.

- CSV (comma-separated values):

```
$context.identity.sourceIp,$context.identity.caller, \
$context.requestTime,$context.httpMethod, \
$context.resourcePath,$context.protocol,$context.status, \
$context.responseLength,$context.requestId
```

The continuation characters (\) are meant as a visual aid and the log format cannot have any line breaks.

Permissions

To enable CloudWatch Logs, you must grant API Gateway proper permissions 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": "*"
        }
    ]
}
```

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](#).

Set up API Logging Using the API Gateway Console

To set up API logging, you must have deployed the API to a stage. You must also have configured [an appropriate CloudWatch Logs role \(p. 384\)](#) ARN for your account.

1. Sign in to the API Gateway console.
2. Choose **Settings** from the primary navigation panel. Type an ARN of an IAM role with appropriate permissions in **CloudWatch log role ARN**. You need to do this once.

3. 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.
4. Choose **Logs** in the **Stage Editor**.
5. To enable execution logging, choose **Enable CloudWatch Logs** under **CloudWatch Settings**. Choose **Error** or **Info** from the drop-down menu. If desired, choose **Enable Detailed CloudWatch Metrics**.
6. To enable access logging, choose **Enable Access Logging** under **Custom Access Logging**. Then type the ARN of a log group in **CloudWatch Group**. Type a log format in **Log Format**. You can choose **CLF**, **JSON**, **XML**, or **CSV** to use one of the provided examples as a guide.
7. Choose **Save Changes**.

Note

You can enable execution logging and access logging independent of each other.

API Gateway is now ready to log requests to your API. You do not need to redeploy the API when you update the stage settings, logs, or stage variables.

Set up Stage Variable for API Deployment

Stage variables are name-value pairs that you can define as configuration attributes associated with a deployment stage of an 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 API. Later, you can reference the URL string using the associated stage variable name from the API setup. This way, 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 Reference \(p. 191\)](#).

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 may want to re-use 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, e.g., `url`, and then assign it a value, e.g., `example.com`. Next, from your

method configuration, set up an HTTP proxy integration, and 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. You can use the AWS Command Line Interface 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
```

The following example assigns API Gateway permission to invoke a Lambda function named `helloWorld` hosted in the US West (Oregon) region of an AWS account on behalf of the API method.

```
arn arn:aws:execute-api:us-west-2:123123123123:bmmuvptwze/*/*GET/hello
```

Here is the same command using the AWS CLI.

```
aws lambda add-permission --function-name arn:aws:lambda:us-east-1:123123123123:function:helloWorld --source-arn arn:aws:execute-api:us-west-2:123123123123:bmmuvptwze/*/*GET/hello --principal apigateway.amazonaws.com --statement-id apigateway-access --action lambda:InvokeFunction
```

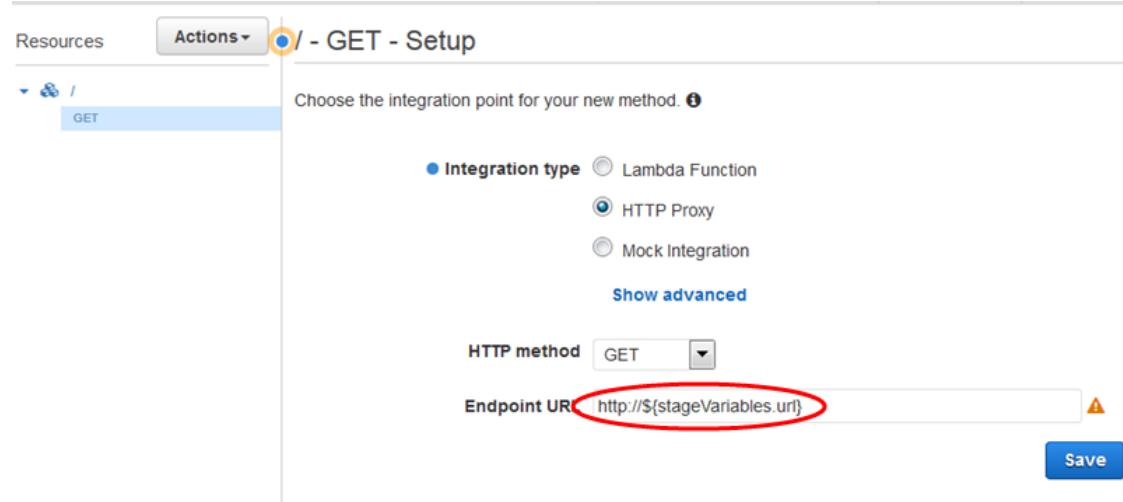
Set Stage Variables Using the Amazon API Gateway Console

In this tutorial, you will learn how to set stage variables for two deployment stages of a sample API, using the Amazon API Gateway console. Before you begin, make sure the following prerequisites are met:

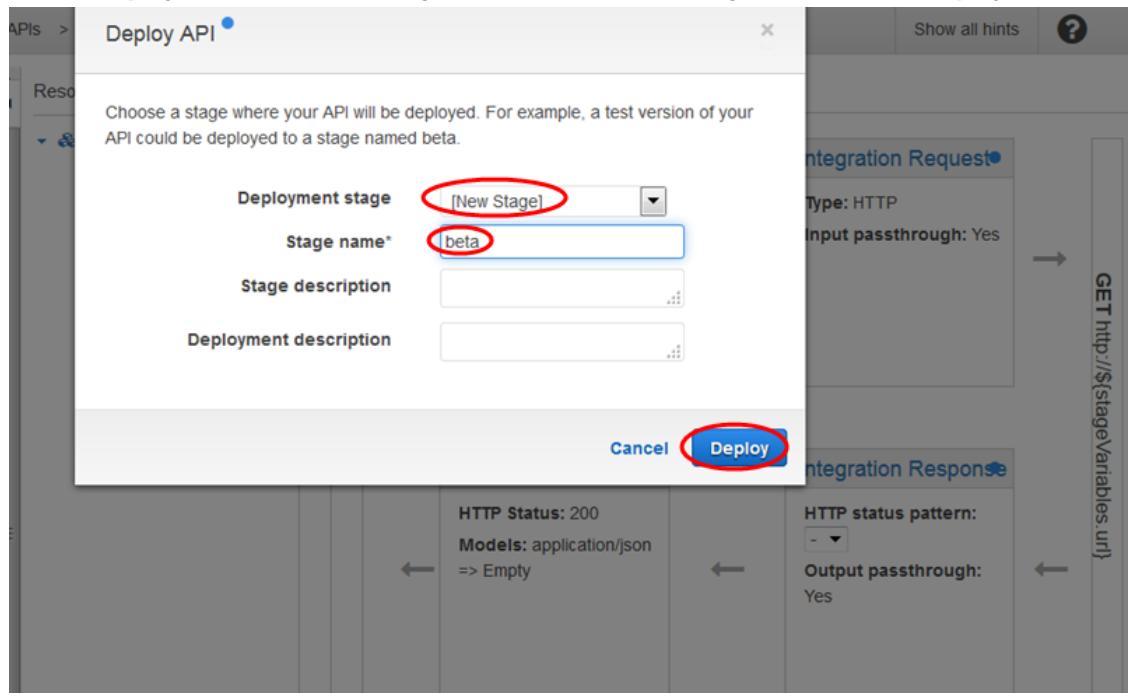
- You must have an API available in API Gateway. Follow the instructions in [Creating an API in Amazon API Gateway \(p. 81\)](#).
- You must have deployed the API at least once. Follow the instructions in [Deploying an API in Amazon API Gateway \(p. 370\)](#).
- You must have created the first stage for a deployed API. Follow the instructions in [Create a New Stage \(p. 373\)](#).

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, create a GET method on the API's root resource, if you have not already done so. Set the HTTP **Endpoint URL** value as "`http://${stageVariables.url}`", and then choose **Save**.



3. Choose **Deploy API**. Choose **New Stage** and enter "beta" for **Stage name**. Choose **Deploy**.



4. In the **beta Stage Editor** panel; choose the **Stage Variables** tab; and then choose **Add Stage Variable**.
5. Enter the "url" string in the **Name** field and the "httpbin.org/get" in the **Value** field. Choose the checkmark icon to save the setting for the stage variable.
6. Repeat the above step to add two more stage variables: **version** and **function**. Set their values as "v-beta" and "HelloWorld", respectively.

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 will expand the given stage variable into an ARN.

7. From the **Stages** navigation pane, choose **Create**. For **Stage name**, type **prod**. Select a recent deployment from **Deployment** and then choose **Create**.

The screenshot shows the 'Create Stage' dialog box. On the left, there's a 'Stages' navigation pane with a 'beta' stage listed. The main area is titled 'Create Stage' with the sub-instruction: 'Create a stage where your APIs will be deployed. For example, a test version of your API could be deployed to a stage named beta.' It has fields for 'Stage name*' (set to 'prod'), 'Stage description' (empty), 'Deployment*' (set to '04-15-2016 10:38'), and 'Deployment Description' (empty). At the bottom right is a large blue 'Create' button.

8. As with the **beta** stage, set the same three stage variables (**url**, **version**, and **function**) to different values ("petstore-demo-endpoint.execute-api.com/petstore/pets", "v-prod", and "HelloEveryone"), respectively.

Use Amazon API Gateway Stage Variables

You can use API Gateway stage variables to access the HTTP and Lambda backends for different API deployment stages and to pass stage-specific configuration metadata into an HTTP backend as a query parameter and into a Lambda function as a payload generated in an input mapping template.

Prerequisites

You must create two stages with a `url` variable set to two different HTTP endpoints: a `function` stage variable assigned to two different Lambda functions, and a `version` stage variable containing stage-specific metadata. Follow the instructions in [Set Stage Variables Using the Amazon API Gateway Console \(p. 386\)](#).

Access an HTTP endpoint through an API with a stage variable

1. In the **Stages** navigation pane, choose **beta**. In **beta Stage Editor**, choose the **Invoke URL** link. 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. Navigating to the URL by choosing the link calls the **beta** stage `GET` method on the root resource. If methods are defined on child resources and not on the root resource itself, choosing the **Invoke URL** link will return 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. From **prod Stage Editor**, choose the **Invoke URL** link. 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 via 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 back end. We will use the `version` stage variable declared in [Set Stage Variables Using the Amazon API Gateway Console \(p. 386\)](#).

1. In the **Resource** navigation pane, choose the **GET** method. To add a query string parameter to the method's URL, in **Method Execution**, choose **Method Request**. Type `version` for the parameter name.

The screenshot shows the 'Method Request' configuration for a GET method. Under 'URL Query String Parameters', there is a table with one row. The 'Name' column contains 'version', which is circled in red. The 'Caching' column has an empty checkbox. There is also a delete icon (cross) in the same row. Below the table, there is a blue '+ Add query string' button. To the right of the table, there are sections for 'HTTP Request Headers' and 'Request Models'.

Name	Caching
version	<input type="checkbox"/>

2. In **Method Execution** choose **Integration Request**. Edit the **Endpoint URL** value to append `?version=${stageVariables.version}` to the previously defined URL value, which, in this case, is also expressed with the `url` stage variable. Choose **Deploy API** to deploy these changes.

The screenshot shows the 'Method Execution' page for a GET request. In the 'Actions' dropdown, 'Deploy API' is highlighted with a red circle. The 'Endpoint URL' field contains the value 'http://\${stageVariables.url}?version=\${stageVariables.version}', with the 'version' part also circled in red.

3. In the **Stages** navigation pane, choose the **beta** stage. From **beta Stage Editor**, verify that the current stage is in the most recent deployment, and then choose the **Invoke URL** link.

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.

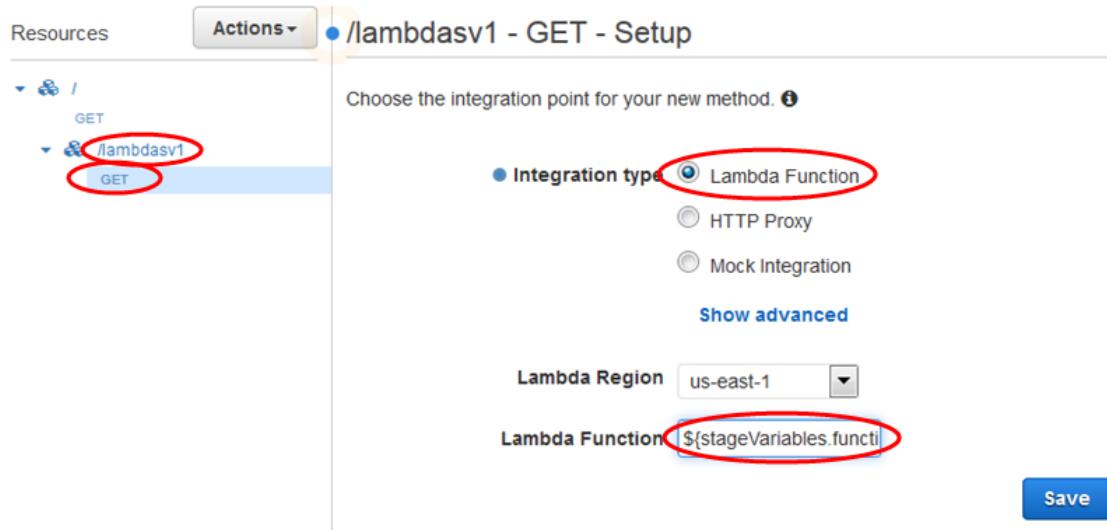
4. The response is shown next. Notice that `v-beta`, assigned to the `version` stage variable, is passed in the backend as the `version` argument.

```
{
  "args": {
    "version": "v-beta"
  },
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "AmazonAPIGateway_h4ah70cvmb"
  },
  "origin": "52.91.42.97",
  "url": "http://httpbin.org/get?version=v-beta"
}
```

Call Lambda function through API with a stage variable

This procedure describes how to use a stage variable to call a Lambda function as a back end of your API. We will use the `function` stage variable declared earlier. For more information, see [Set Stage Variables Using the Amazon API Gateway Console \(p. 386\)](#).

1. In the **Resources** pane, create a **/lambdasv1** child resource under the root directory, and then create a GET method on the child resource. Set the **Integration type** to **Lambda Function**, and in **Lambda Function**, type `#{stageVariables.function}`. Choose **Save**.

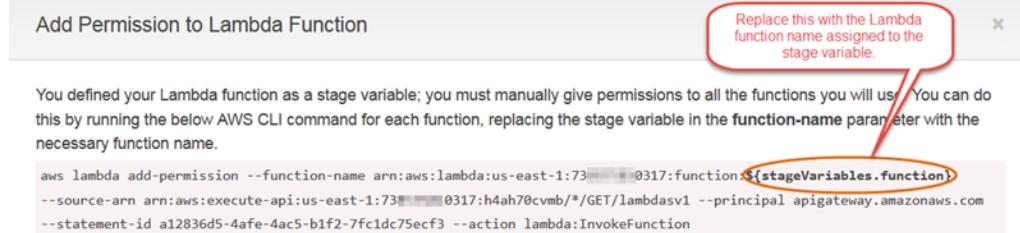


Tip

When prompted with **Add Permission to Lambda Function**, make a note of the AWS CLI command before choosing **OK**. You must run the command on each Lambda function that is or will be assigned to the function stage variable for each of the newly created API methods. For example, if the `#{stageVariables.function}` value is `HelloWorld` and you have not added permission to this function yet, you must 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/lambdasv1 --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. Make sure to replace `#{stageVariables.function}` with the Lambda function name that is assigned to the stage variable.



2. Deploy the API to available stages.
3. In the **Stages** navigation pane, choose the **beta** stage. Verify that your most recent deployment is in **beta Stage Editor**. Copy the **Invoke URL** link, paste it into the address bar of your browser, and append `/lambdasv1` to that URL. This calls the underlying Lambda function through the GET method on the **LambdaSv1** child resource of the API.

Note

Your HelloWorld Lambda function implements the following code.

```
exports.handler = function(event, context, callback) {
  if (event.version)
    callback(null, 'Hello, World! (' + event.version + ')');
  else
    callback(null, "Hello, world! (v-unknown)");
};
```

This implementation results in the following response.

```
"Hello, world! (v-unknown)"
```

Pass stage-specific metadata to a Lambda function via a stage variable

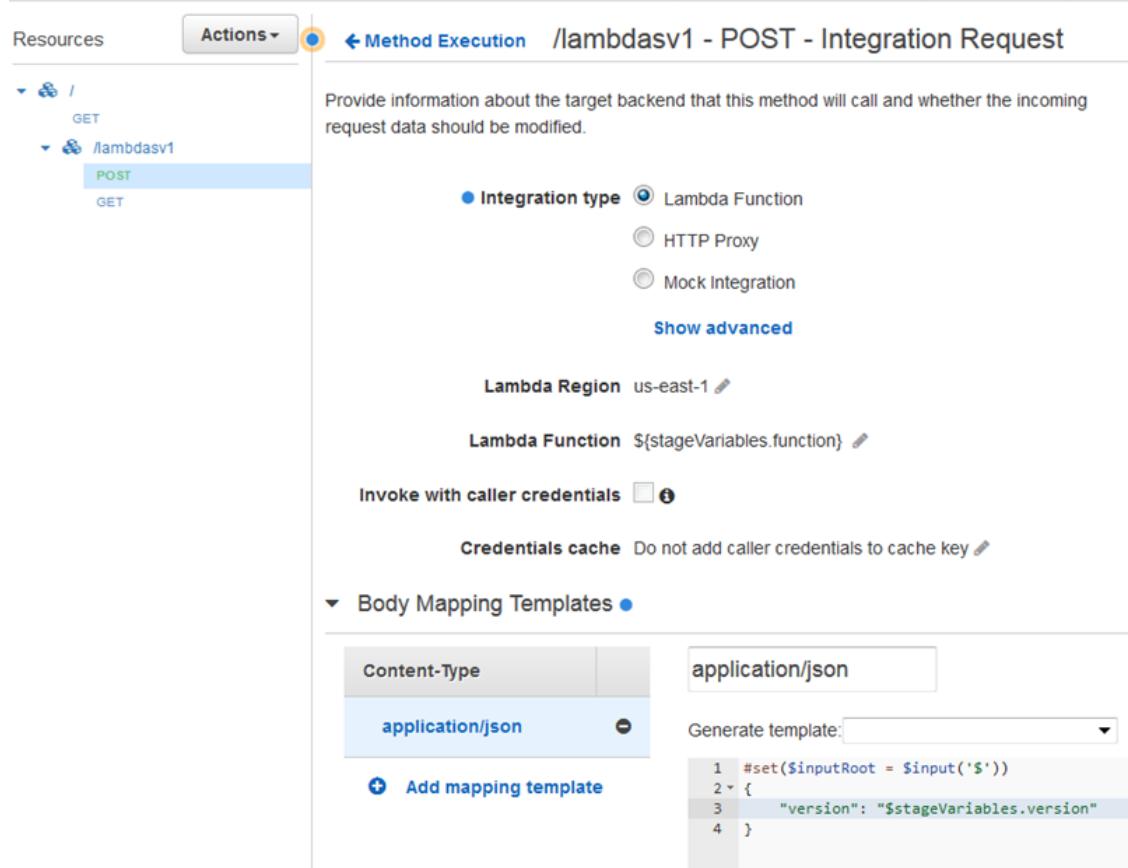
This procedure describes how to use a stage variable to pass stage-specific configuration metadata into a Lambda function. We will use a `POST` method and an input mapping template to generate payload using the `version` stage variable declared earlier.

1. In the **Resources** pane, choose the `/lambdasv1` child resource. Create a `POST` method on the child resource, set the **Integration type** to **Lambda Function**, and type `#{stageVariables.function}` in **Lambda Function**. Choose **Save**.

Tip

This step is similar to the step we used to create the `GET` method. For more information, see [Call Lambda function through API with a stage variable \(p. 390\)](#).

2. From the **/Method Execution** pane, choose **Integration Request**. In the **Integration Request** pane, expand **Mapping Templates**, and then choose **Add mapping template** to add a template for the `application/json` content-type, as shown in the following.



The screenshot shows the 'Method Execution' configuration for a POST request to the '/lambdasv1' endpoint. The 'Integration type' is set to 'Lambda Function', with the Lambda function name being `\${stageVariables.function}`. Under 'Body Mapping Templates', for the 'Content-Type' `application/json`, there is a mapping template defined:

```

1 #set($inputRoot = $input('$'))
2 {
3     "version": "$stageVariables.version"
4 }

```

Note

In a mapping template, a stage variable must be referenced within quotes (as in `"$stageVariables.version"` or `"${stageVariables.version}"`), whereas elsewhere it must be referenced without quotes (as in `${stageVariables.function}`).

3. Deploy the API to available stages.
4. In the **Stages** navigation pane, choose **beta**. In **beta Stage Editor**, verify that the current stage has the most recent deployment. Copy the **Invoke URL** link, paste it into the URL input field of a REST API client, append `/lambdasv1` to that URL, and then submit a **POST** request to the underlying Lambda function.

Note

You will get the following response.

```
"Hello, world! (v-beta)"
```

To summarize, we have demonstrated how to use API Gateway stage variables to target different HTTP and Lambda back ends for different stages of API deployment. In addition, we also showed how to use the stage variables to pass stage-specific configuration data into HTTP and Lambda back ends. Together, these procedures demonstrate the versatility of the API Gateway stage variables in managing API development.

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, e.g., `http://${stageVariables.<variable_name>}`
- A full domain: e.g., `http://${stageVariables.<variable_name>}/resource/operation`
- A subdomain: e.g., `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- A path, e.g., `http://example.com/${stageVariables.<variable_name>}/bar`
- A query string, e.g., `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:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

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 Tags for an API Stage in API Gateway

Tags are metadata that you assign to your AWS resources. They are commonly used for tracking resource usage by custom-defined categories, which provides a simple mechanism to separate distinct organizational units within a single AWS account. A tag consists of a key-value pair. For more information on how to use tags, see [AWS Tagging Strategy](#).

In API Gateway, you can assign tags to an API stage for managing cost allocation for request invocation and caching that are associated with the stage. For example, when you add the tag `Department:Sales` to an API stage, it shows up in AWS Billing and Cost Management as a cost allocation tag. After a tag is activated, you can use it to filter costs and usage by using [Cost Explorer](#) in the [AWS Billing and Cost Management](#) console.

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.

Topics

- [Set up Tags for an API Stage Using the API Gateway Console \(p. 395\)](#)
- [Set up Tags for an API Stage Using the API Gateway REST API \(p. 395\)](#)
- [Tag Restrictions \(p. 397\)](#)

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 **Stage Editor**, choose the **Settings** tab.
5. Under the **Tags** section, choose **Add Stage Tag**. Type a tag key (for example, `Department`) in the **Key** column, and type a tag value (for example, `Sales`) in the **Value** column. Choose the checkmark icon to save the tag.
6. If needed, repeat Step 5 to add more tags to the API stage. The maximum number of tags per stage is 50.
7. To remove an existing tag from the stage, choose the trash bin icon next to the selected tag.
8. Choose **Save Changes** to finish setting up the stage tags.

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.

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%2Frestapis%2Fm5zr3vnks7%2Fstages%2Ftest`).

```
PUT /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%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%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%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"
  }
}
```

Tag Restrictions

The following restrictions apply to tags for API Gateway resources:

- Tags are applicable to the [Stage](#) resource only.
- The maximum number of tags per stage is 50.
- The maximum tag key length is 128 Unicode characters in UTF-8.
- The maximum tag value length is 256 Unicode characters in UTF-8.
- Tag keys and values are case sensitive.
- The valid character set is [a-zA-Z+-_.:/] for tag keys and values.
- Tag keys and values can't start with aws:..

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 as a canary release 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 \(p. 398\)](#)
- [Create a Canary Release Deployment \(p. 399\)](#)
- [Update a Canary Release \(p. 406\)](#)
- [Promote a Canary Release \(p. 408\)](#)
- [Disable a Canary Release \(p. 412\)](#)

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, AWS CLI, an AWS SDK, and the API Gateway REST API.

Topics

- [Create a Canary Deployment Using the API Gateway Console \(p. 399\)](#)
- [Create a Canary Deployment Using the AWS CLI \(p. 400\)](#)
- [Create a Canary Deployment Using the API Gateway API \(p. 403\)](#)

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 API or create a new API.
3. Change the API, if necessary, or set up desired API methods and integrations.
4. Choose **Deploy API** from the **Actions** drop-down menu. 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.

5. To enable API caching, choose the **Settings** tab in **Stage Editor** and follow the on-screen instructions. For more information, see [the section called "Enable API Caching" \(p. 378\)](#).
6. To set stage variables, choose the **Stage Variables** tab in **Stage Editor** and follow the on-screen instructions to add or modify stage variables. For more information, see [the section called "Set up Stage Variables" \(p. 385\)](#).
7. To configure execution or access logging, choose the **Logs** tab in **Stage Editor** and follow the on-screen instructions. For more information, see [Set up API Logging in API Gateway \(p. 382\)](#).
8. In **Stage Editor**, choose the **Canary** tab and then choose **Create Canary**.
9. Under the **Stage's Request Distribution** section, choose the pencil icon next to **Percentage of requests to Canary** and type a number (for example, `5.0`) in the input text field. Choose the check mark icon to save the setting.

10. If needed, choose **Add Stage Variables** to add them under the **Canary Stage Variables** section to override existing stage variables or add new stage variables for the canary release.
11. If desired, choose **Enable use of stage cache** to enable caching for the canary release and save your choice. The cache is not available for the canary release until API caching is enabled.

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** from the **Actions** drop-down menu next to the **Resources** list.
2. In **Deploy API**, choose the now canary-enabled stage from the **Deployment stage** drop-down list.
3. Optionally, type a description in **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 described in [To create the initial canary release deployment \(p. 399\)](#).

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 4wk1k4onj3
--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
--canary-settings '{ \
    "percentTraffic":10.5, \
    "useStageCache":false, \
    "stageVariableOverrides":{ \
        "sv1":"val2", \
        "sv2":"val3" \
    } \
}' \
--rest-api-id 4wk1k4onj3 \
--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 4wk1k4onj3 --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 4wk1k4onj3 \
--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 \
--patch-operations '[{ \
    "op":"replace", \
    "path":"/canarySettings/percentTraffic", \
    "value":"10.5" \
},{ \
    "op":"replace", \
    "path":"/canarySettings/useStageCache", \
    "value":"false" \
},{ \
    "op":"replace", \
    "path":"/canarySettings/stageVariableOverrides/sv1", \
    "value":"val2" \
},{ \
    "op":"replace", \
    "path":"/canarySettings/stageVariableOverrides/sv2", \
    "value":"val3" \
}]' \
--rest-api-id 4wk1k4onj3 \
--stage-name beta
```

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.

Create a Canary Deployment Using the API Gateway API

To use the API Gateway REST API to deploy your API as a canary release, call [deployment:create](#) as follows:

```
POST /restapis/fugvwdxtri/deployments HTTP/1.1
Content-Type: application/json
Host: apigateway.us-east-1.amazonaws.com
X-Amz-Date: 20171103T175605Z
Authorization: AWS4-HMAC-SHA256 Credential={SECRET_ACCESS_KEY}/20160603/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature={SIGV4_SIGNATURE}

{
  "stageName" : "prod",
  "stageDescription" : "Production stage",
  "description" : "Production deployment with canary",
  "variables" : {
    "sv1" : "val1"
  },
  "canarySettings": {
    "percentTraffic": 10.5,
    "useStageCache": false,
    "stageVariableOverrides": {
      "sv1": "val2",
      "sv2": "val3"
    }
  }
}
```

If successful, and if this is the first time to deploy the API to the stage, you get a brand new canary release deployment (nfcn0x):

```
{
  "_links": {
    ...
  },
  "createdDate": "2017-11-22T00:54:28Z",
  "description": "Production deployment with canary",
  "id": "nfcn0x"
}
```

In this deployment, both the stage and the canary have the same deploymentId. That is, they both reference the same API version.

In any subsequent API deployments to the same stage, you must always specify canarySettings as an input, until the canary is disabled on that stage. For example, when you call the previous deployment:create request the second time, you get a new deployment (eh1sby) as the result:

```
{
    "_links": {
        ...
    },
    "createdDate": "2017-11-22T01:24:23Z",
    "description": "Production deployment with canary",
    "id": "eh1sby"
}
```

The newer deploymentId value is set on the canarySettings and the canary represents the new API version, while the initial deploymentId remains associated with the stage that represents the initial API version. You can verify this by calling the [GET /restapis/fugvwdxtri/stages/prod](#) request and examining the successful response payload:

```
{
    "_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": {
        "*/*": {
            "dataTraceEnabled": true,
            "throttlingRateLimit": 10000,
            "cacheTtlInSeconds": 300,
            "cachingEnabled": false,
            "requireAuthorizationForCacheControl": true,
            "metricsEnabled": true,
            "loggingLevel": "INFO",
            "unauthorizedCacheControlHeaderStrategy": "SUCCEED_WITH_RESPONSE_HEADER",
            "throttlingBurstLimit": 5000,
            "cacheDataEncrypted": false
        }
    },
    "stageName": "canary",
    "variables": {
        "sv1": "val1"
    }
}
```

For a regular production deployment without a canary enabled on the associated stage, you can turn the deployment into a canary release deployment by enabling the canary on the stage. To do this, call [stage:update](#), as shown in the following, assuming the original deployment ID is ghdx4w:

```
PATCH /restapis/4wk1k4onj3/stages/prod HTTP/1.1
Host: apigateway.us-east-1.amazonaws.com
Content-Type: application/json
```

```
X-Amz-Date: 20171121T232431Z
Authorization: AWS4-HMAC-SHA256 Credential={SECRET_ACCESS_KEY}/20171121/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature={SIGV4_SIGNATURE}

{
  "patchOperations": [
    {
      "op": "replace",
      "path": "/canarySettings/deploymentId",
      "value": "ghdx4w"
    },
    {
      "op": "replace",
      "path": "/canarySettings/percentTraffic",
      "value": "15.0"
    }
  ]
}
```

Because the original deployment is without a canary release, we set the `/canarySettings/deploymentId` value to the `deploymentId` ("`ghdx4w`") associated with the deployment stage.

The successful response returns a payload similar to the following:

```
{
  "_links": {
    ...
  },
  "accessLogSettings": {
    ...
  },
  "cacheClusterEnabled": false,
  "cacheClusterStatus": "NOT_AVAILABLE",
  "canarySettings": {
    "deploymentId": "ghdx4w",
    "useStageCache": false,
    "stageVariableOverrides": null,
    "percentTraffic": 15
  },
  "createdDate": "2017-11-20T04:42:19Z",
  "deploymentId": "ghdx4w",
  "lastUpdatedDate": "2017-11-21T23:24:31Z",
  "methodSettings": {
    "*/*": {
      "dataTraceEnabled": true,
      "throttlingRateLimit": 10000,
      "cacheTtlInSeconds": 300,
      "cachingEnabled": false,
      "requireAuthorizationForCacheControl": true,
      "metricsEnabled": true,
      "loggingLevel": "INFO",
      "unauthorizedCacheControlHeaderStrategy": "SUCCEED_WITH_RESPONSE_HEADER",
      "throttlingBurstLimit": 5000,
      "cacheDataEncrypted": false
    }
  },
  "stageName": "prod"
}
```

With a canary enabled on the stage, the deployment becomes a canary release deployment. The stage cannot be associated with any non-canary deployment until the canary settings are removed from the stage.

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, an AWS SDK, and the API Gateway REST API's [stage:update](#) link-relation.

Topics

- [Update a Canary Release Using the API Gateway Console \(p. 406\)](#)
- [Update a Canary Release Using the AWS CLI \(p. 406\)](#)
- [Update a Canary Release Using the API Gateway REST API \(p. 407\)](#)

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:

1. Sign in to the API Gateway console and choose an existing API in the primary navigation pane.
2. Choose **Stages** under the API and then choose an existing stage under the **Stages** list to open the **Stage Editor**.
3. Choose the **Canary** tab in the **Stage Editor**.
4. Update **Percentage of requests directed to Canary** by increasing or decreasing the percentage number between 0.0 and 100.0, inclusive.
5. Update **Canary Stage Variables**, including adding, removing, or modifying a desired stage variable.
6. Update the **Enable use of stage cache** option by selecting or clearing the check box.
7. Save the changes.

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/percentTraffic,value=25.0
```

```
--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": "/canary/percentTraffic",
    "value": "20.0"
}, {
    "op": "replace",
    "path": "/canary/useStageCache",
    "value": "true"
}, {
    "op": "remove",
    "path": "/canarySettings/stageVariableOverrides/var1"
}, {
    "op": "replace",
    "path": "/canarySettings/stageVariableOverrides/newVar",
    "value": "newVal"
}, {
    "op": "replace",
    "path": "/canarySettings/stageVariableOverrides/var2",
    "value": "val4"
}]'
```

Update a Canary Release Using the API Gateway REST API

To enable or disable the use of a stage cache for the canary, call [stage:update](#) as follows:

```
PATCH /restapis/{rest-api-id}/stages/{stage-name}

{
    "patchOperations": [
        {
            "op": "replace",
            "path": "/canarySettings/useStageCache",
            "value": "true"
        }
    ]
}
```

To adjust the canary traffic percentage, call [stage:update](#) to replace the /canarySettings/percentTraffic value on the [stage](#).

```
PATCH /restapis/{rest-api-id}/stages/{stage-name}

{
```

```
    "patchOperations": [
        {
            "op": "replace",
            "path": "/canarySettings/percentTraffic",
            "value": "25.0"
        }
    ]
}
```

To update canary stage variables, including adding, changing, or removing the canary stage variable, use the following example:

```
PATCH /restapis/{rest-api-id}/stages/{stage-name}

{
    "patchOperations": [
        {
            "op": "replace",
            "path": "/canarySettings/stageVariableOverrides/newVar",
            "value": "newVal"
        },
        {
            "op": "replace",
            "path": "/canarySettings/stageVariableOverrides/var2",
            "value": "val4"
        },
        {
            "op": "remove",
            "path": "/canary/overriddenStageVariables/var1",
        }
    ]
}
```

You can combine all of the above operations into a single `PATCH` request.

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 \(p. 408\)](#)
- [Promote a Canary Release Using the AWS CLI \(p. 409\)](#)
- [Promote a Canary Release Using the API Gateway REST API \(p. 410\)](#)

Promote a Canary Release Using the API Gateway Console

To use the API Gateway console to promote a canary release deployment, do the following:

1. Sign in to the API Gateway console and choose an existing API in the primary navigation pane.

2. Choose **Stages** under the API and then choose an existing stage under the **Stages** list to open the **Stage Editor**.
3. Choose the **Canary** tab in the **Stage Editor**.
4. Choose **Promote Canary**.
5. Confirm changes to be made and choose **Update**.

After the promotion, the production release references the same API version (**deploymentId**) as the canary release. You can verify this using the AWS CLI or API Gateway REST API. For example, see [the section called "Promote a Canary Release Using the AWS CLI" \(p. 409\)](#) or [the section called "Promote a Canary Release Using the API Gateway REST API" \(p. 410\)](#).

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": "/canary/overriddenStageVariables" \
        "path": "/variables",
    }, {
        "op": "copy",
        "from": "/canary/deploymentId",
        "path": "/deploymentId"
    }]'
```

After the promotion, the stage now looks like this:

```
{
    "_links": {
        ...
    },
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "canarySettings": {
        "deploymentId": "ehlsby",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        },
        "percentTraffic": 0
    },
    "createdDate": "2017-11-20T04:42:19Z",
    "deploymentId": "ehlsby",
    "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 “Disable a Canary Release” \(p. 412\)](#).

Promote a Canary Release Using the API Gateway REST API

To promote a canary release to the production release using the API Gateway REST API, call the `stage:update` request 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 stage:update request to promote it:

```

PATCH /restapis/4wk1k4onj3/stages/prod HTTP/1.1
Host: apigateway.us-east-1.amazonaws.com
Content-Type: application/json
X-Amz-Date: 20171121T232431Z
Authorization: AWS4-HMAC-SHA256 Credential={SECRET_ACCESS_KEY}/20171121/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature={SIGV4_SIGNATURE}

{
    "patchOperations": [
        {
            "op": "copy",
            "path": "/deploymentId",
            "from": "/canarySettings/deploymentId"
        },
        {
            "op": "replace",
            "path": "/canarySettings/percentTraffic",
            "value": "0.0"
        },
        {
            "op": "copy",
            "path": "/variables",
            "from": "/canarySettings/stageVariableOverrides"
        }
    ]
}

```

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 “Disable a Canary Release” \(p. 412\)](#).

Disable a Canary Release

To disable 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, AWS CLI, an AWS SDK, or the API Gateway REST API.

Topics

- [Disable a Canary Release Using the API Gateway Console \(p. 412\)](#)
- [Disable a Canary Release Using the AWS CLI \(p. 413\)](#)
- [Disable a Canary Release Using the API Gateway REST API \(p. 413\)](#)

Disable a Canary Release Using the API Gateway Console

To use the API Gateway console to disable a canary release deployment, use the following steps:

1. Sign in to the API Gateway console and choose an existing API in the primary navigation pane.
2. Choose **Stages** under the API and then choose an existing stage under the **Stages** list to open the **Stage Editor**.
3. Choose the **Canary** tab in the **Stage Editor**.
4. Choose **Delete Canary**.
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 or the API Gateway REST API. For example, see [the section called](#)

["Disable a Canary Release Using the AWS CLI" \(p. 413\)](#) or the section called ["Disable a Canary Release Using the API Gateway REST API" \(p. 413\)](#).

Disable a Canary Release Using the AWS CLI

To use the AWS CLI to disable a canary release deployment, call the `update-stage` command as follows:

```
aws apigateway update-stage \  
  --rest-api-id 4wk1k4onj3 \  
  --stage-name canary \  
  --patch-operations '[{"op":"remove", "path":"/canarySettings"}]
```

The successful response returns an output similar to this:

```
{  
  "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.

Disable a Canary Release Using the API Gateway REST API

To use the API Gateway REST API to disable a canary release deployment, make the `stage:update` request as follows:

```
PATCH /restapis/4wk1k4onj3/stages/prod HTTP/1.1  
Host: apigateway.us-east-1.amazonaws.com  
Content-Type: application/json  
X-Amz-Date: 20171121T230325Z  
Authorization: AWS4-HMAC-SHA256 Credential={SECRET_ACCESS_KEY}/20171121/us-  
east-1/apigateway/aws4_request, SignedHeaders=content-type;host;x-amz-date,  
Signature={SIGV4_SIGNATURE}  
  
{  
  "patchOperations": [  
    {  
      "op": "remove",  
      "path": "/canarySettings"  
    }  
  ]  
}
```

The successful response returns an output similar to this:

```
{
```

```
"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.

Export an API from API Gateway

Once you created and configured an API in API Gateway, using the API Gateway console or otherwise, you can export it to a Swagger file using the API Gateway Export API, which is part of the Amazon API Gateway Control Service. You have options to include the API Gateway integration extensions, as well as the [Postman](#) extensions, in the exported Swagger definition file.

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 an API

With the Export API, you export an existing 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:

```
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 [Making HTTP Requests](#).

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.

Download API Swagger Definition in JSON

To export and download an API in Swagger definitions in JSON format:

```
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 API Swagger Definition in YAML

To export and download an API in Swagger definitions in YAML format:

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

Download API Swagger Definition with Postman Extensions in JSON

To export and download an API in Swagger definitions with the Postman in JSON format:

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

Download API Swagger Definition with API Gateway Integration in YAML

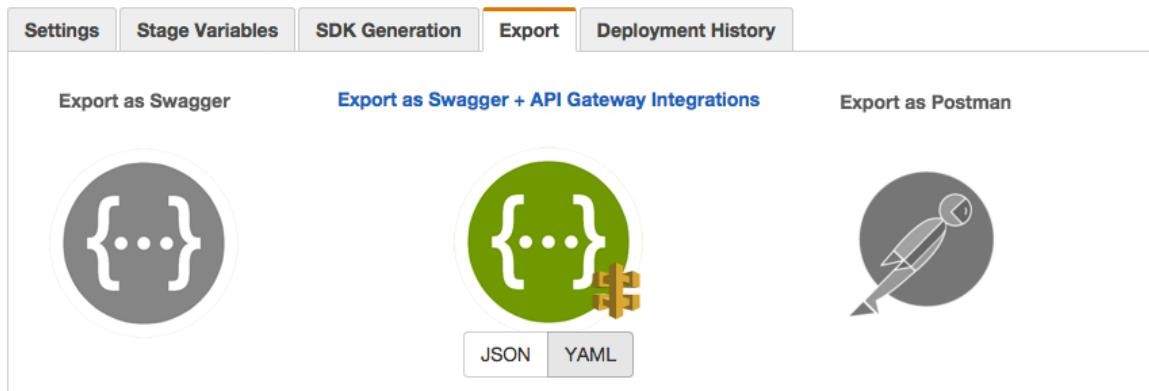
To export and download an API in Swagger definitions with API Gateway integration in YAML format:

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?extensions=integrations
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

Export API Using the API Gateway Console

After [deploying your API to a stage \(p. 372\)](#), you can proceed to export the API in the stage to a Swagger file using the API Gateway console.

From the **stage configuration** page in the API Gateway console, choose the **Export** tab and then one of the available options (**Export as Swagger**, **Export as Swagger + API Gateway Integrations** and **Export as Postman**) to download your API's Swagger definition.



Generate SDK of an API

To call your 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 and 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 \(p. 422\)](#), which exposes this [Simple Calculator \(p. 420\)](#) Lambda function.

Before proceeding, create or import the API and deploy it at least once in API Gateway. For instructions, see [Deploying an API in Amazon API Gateway \(p. 370\)](#).

Topics

- [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#)
- [Generate SDKs for an API Using AWS CLI Commands \(p. 419\)](#)
- [Simple Calculator Lambda Function \(p. 420\)](#)
- [Simple Calculator API in API Gateway \(p. 422\)](#)
- [Simple Calculator API Swagger Definition \(p. 427\)](#)

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 \(p. 427\)](#) 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 \(p. 422\)](#).

Topics

- [Generate the Java SDK of an API \(p. 417\)](#)
- [Generate the Android SDK of an API \(p. 417\)](#)
- [Generate the iOS SDK of an API \(p. 418\)](#)

- [Generate the JavaScript SDK of an API \(p. 418\)](#)
- [Generate the Ruby SDK of an API \(p. 419\)](#)

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. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose **Java** 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**, type `maven` or `gradle` to specify the build system.
 - d. For **Java Group Id**, type a group identifier for your SDK project. For example, `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**, type an artifact identifier for your SDK project. For example, `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**, type 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**, type the license text of your source code, if any.
5. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.
6. Follow the instructions in [Use a Java SDK Generated by API Gateway \(p. 460\)](#) 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. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose the Android platform.
 - a. For **Group ID**, type the unique identifier for the corresponding project. This is used in the `pom.xml` file (for example, `com.mycompany`).
 - b. For **Invoker package**, type the namespace for the generated client classes (for example, `com.mycompany.clientsdk`).

- c. For **Artifact ID**, type 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**, type 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**).
5. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.
 6. Follow the instructions in [Use an Android SDK Generated by API Gateway \(p. 463\)](#) 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. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose the **iOS (Objective-C) or iOS (Swift)** platform.
 - 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 \(p. 422\)](#) 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 in Objective-C or Swift \(p. 469\)](#).

5. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.
6. Follow the instructions in [Use iOS SDK Generated by API Gateway in Objective-C or Swift \(p. 469\)](#) 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 an 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. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose **JavaScript**.
5. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.
6. Follow the instructions in [Use a JavaScript SDK Generated by API Gateway \(p. 465\)](#) 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. In the box that contains the name of the API for the stage, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose **Ruby**.
 - 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`.
5. Choose **Generate SDK**, and then follow the on-screen directions to download the SDK generated by API Gateway.
6. Follow the instructions in [Use a Ruby SDK Generated by API Gateway \(p. 466\)](#) 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 AWS CLI \(p. 419\)](#)
- [Generate and Download the JavaScript SDK Using AWS CLI \(p. 420\)](#)
- [Generate and Download the Ruby SDK Using AWS CLI \(p. 420\)](#)

Generate and Download the Java for Android SDK Using 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 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 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 API through Generated SDKs \(p. 459\)](#).

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 \(p. 420\)](#)
- [Simple Calculator Lambda Function Output Format \(p. 421\)](#)
- [Simple Calculator Lambda Function Implementation \(p. 421\)](#)
- [Create the Simple Calculator Lambda Function \(p. 421\)](#)

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:

```
console.log('Loading the Calc function');

exports.handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    if (event.a === undefined || event.b === undefined || event.op === undefined) {
        callback("400 Invalid Input");
    }

    var res = {};
    res.a = Number(event.a);
    res.b = Number(event.b);
    res.op = event.op;

    if (isNaN(event.a) || isNaN(event.b)) {
        callback("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":
            res.c = res.b==0 ? NaN : Number(event.a) / Number(event.b);
            break;
        default:
            callback("400 Invalid Operator");
            break;
    }
    callback(null, res);
};
```

Create the Simple Calculator Lambda Function

You can use the AWS Lambda console at <https://console.aws.amazon.com/lambda/> to create the function, pasting the above code listing into the online code editor as follows.

Lambda > Functions > Calc

Qualifiers ▾ Test Actions ▾

Code Configuration Triggers Monitoring

Code entry type Edit code inline ▾

```
1  console.log('Loading the Calc function');
2
3  exports.handler = function(event, context) {
4      console.log('Received event:', JSON.stringify(event, null, 2));
5      if (event.a === undefined || event.b === undefined || event.op === undefined) {
6          context.fail("400 Invalid Input");
7      }
8
9      var res = {};
10     res.a = Number(event.a);
11     res.b = Number(event.b);
12     res.op = event.op;
13 }
```

Simple Calculator API in API Gateway

Our simple calculator API exposes three methods (GET, POST, GET) to invoke the [Simple Calculator Lambda Function \(p. 420\)](#) (Calc). A graphical representation of this API is shown as follows:

The screenshot shows the AWS API Gateway console. On the left, there's a sidebar with links for APIs, SimpleCalc, Resources, Stages, Authorizers, Models, and Dashboard. Below that is a section for Usage Plans, API Keys, Custom Domain Names, Client Certificates, and Settings. The main content area shows a resource path / with three methods: GET, POST, and a nested resource /{a} with methods ANY, ANY, and /{op} with method 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 an API in Amazon API Gateway \(p. 81\)](#). If you're new to API Gateway, see [Build an API Gateway API with Lambda Integration \(p. 18\)](#) first.

Create Models for Input and Output

To specify strongly typed input in the SDK, we create an `Input` model for the API:

Provide a name, content type, and a schema for your model. Models use [JSON schema](#).

Model schema*

```

1 - {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "type": "object",
4   "properties": {
5     "a": {"type": "number"},
6     "b": {"type": "number"},
7     "op": {"type": "string"}
8   },
9   "title": "Input"
10 }

```

* Required Cancel Create model

Similarly, to describe the response body data type, we create the following models in the API Gateway:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "c": {"type": "number"}
  },
  "title": "Output"
}
```

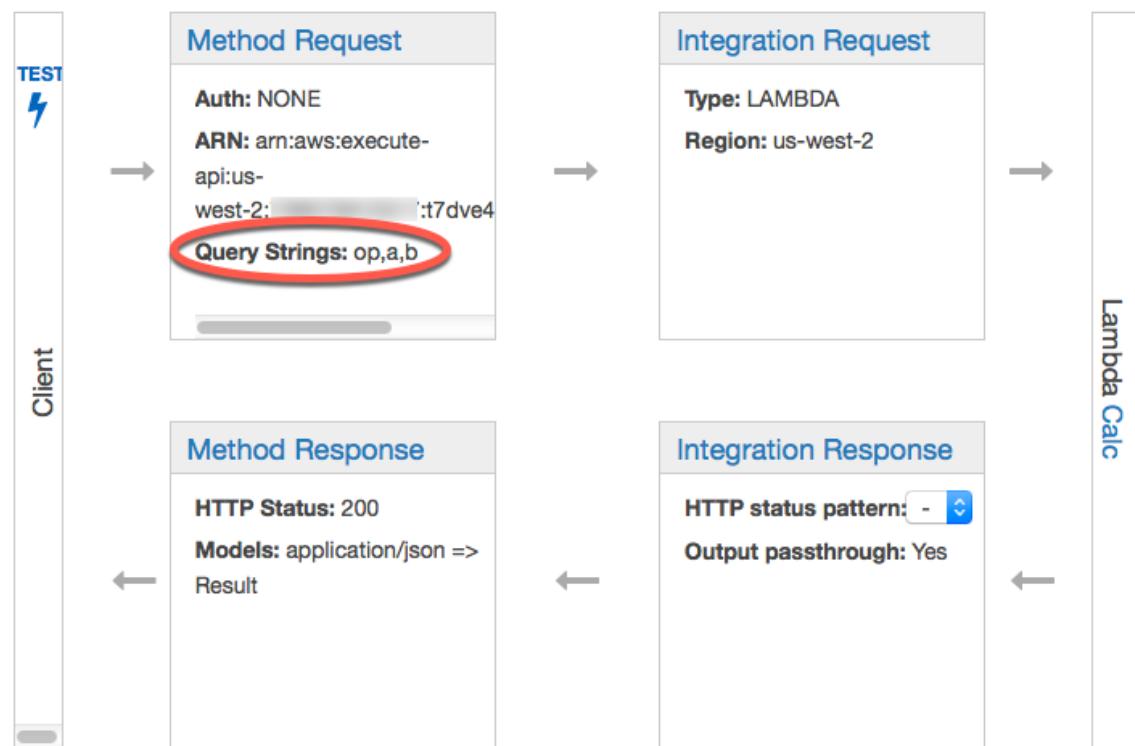
and

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "input": {
      "$ref": "https://apigateway.amazonaws.com/restapis/t7dve4zn36/models/Input"
    },
    "output": {
      "$ref": "https://apigateway.amazonaws.com/restapis/t7dve4zn36/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**:

/ - GET - Method Execution



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.

[← Method Execution](#) / - POST - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Authorization Settings

Authorization: NONE  

API Key Required: false 

▶ URL Query String Parameters

▶ HTTP Request Headers

▼ Request Models [Create a Model](#)

Content type	Model name	
application/json	 Input 	

 [Add model](#)

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.

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.

[Method Execution](#) /{a}/{b}/{op} - GET - Method Response

Provide information about this method's response types, their headers and content types.

HTTP Status	
▼ 200	✖

Response Headers for 200

Name	
No headers	
+ Add Header	

Response Models for 200

Create a model	
Content type	Models
application/json	Result
	 
+ Add Response Model	

[+ Add Response](#)

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.

In addition, you can also create and set up the API following the [Swagger API definitions \(p. 427\)](#).

Simple Calculator API Swagger Definition

The following is the Swagger 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 \(p. 420\)](#) 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](#).

```
{
  "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\": \"$inputRoot.a\"\n    ,\n    \"b\": \"$inputRoot.b\"\n    ,\n    \"op\": \"$inputRoot.op\"\n}\n,\n    \"output\": {\n        \"c\": \"$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 $context.httpMethod\n$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"
                }
            ]
        }
    }
}

```

```

        "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"
    }
}
```

Set up Custom Domain Name for an API in API Gateway

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* is specified by you when creating the API and *stage* is specified by you when deploying the API.

The host name portion of the URL (i.e., *api-id*.execute-api.*region*.amazonaws.com) refers to an API endpoint, which can be edge-optimized or regional. The default API endpoint can be difficult to recall and not user-friendly. To provide a simpler and more intuitive URL for your API users, you can set up a custom domain name (e.g., *api.example.com*) as the API's host name and choose a base path (e.g., *myservice*) to map the alternative URL to this API. The more user-friendly API base URL now becomes:

```
https://api.example.com/myservice
```

If you do not set any base mapping under a custom domain name, the resulting API's base URL is the same as the custom domain (e.g., https://*api.example.com*.) In this case, the custom domain name cannot support more than one API.

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 for API requests 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.

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 for API requests bound for the custom domain name to be routed to API Gateway through the mapped regional API endpoint. 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 \(p. 439\)](#).

To set up an edge-optimized custom domain name or to update its certificate, you must have a permission to update CloudFront distributions. You can do so by attaching the following IAM policy statement to an IAM user, group or role in your account:

```
{  
    "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 host name, 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.

To provide a certificate for a regional 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: create in or import into ACM a certificate for the given domain name, 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.

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 (aka, zone apex) of a registered Internet domain.

After a custom domain name is created in API Gateway, you must create or update your domain name service (DNS) provider's resource record to map the edge-optimized custom domain name to its CloudFront distribution domain name or to map the regional custom domain name to its regional 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](#).

This section describes how to use ACM to create an SSL/TLS certificate for a custom domain name, to set up the custom domain name for an API, to associate a specific API with a base path under the custom domain name, and to renew (aka rotate) an expiring certificate that was imported into ACM for the custom domain name.

Topics

- [Get Certificates Ready in AWS Certificate Manager \(p. 434\)](#)
- [How to Create an Edge-Optimized Custom Domain Name \(p. 436\)](#)
- [Set up a Custom Domain Name for a Regional API in API Gateway \(p. 442\)](#)
- [Migrate a Custom Domain Name to a Different API Endpoint \(p. 447\)](#)

Get 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.

To get a certificate for a given domain name issued by or imported into ACM

1. Register your Internet domain; e.g., `myDomain.com`. You can use either [Amazon Route 53](#) or a third-party accredited domain registrar. For a list of such registrar, see [Accredited Registrar Directory](#) at the ICANN website.
2. 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. Type a custom domain name for your API; e.g., `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**, type 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**, type or paste your PEM-formatted certificate's private key. The following shows an abbreviated example of such a key.

```
-----BEGIN RSA PRIVATE KEY-----  
EXAMPLEBAAKCAQEAA2Qb3LDHD7StY7Wj6U2/opV6Xu37qUCCkeDWhwpZMYJ9/nETO  
...  
1qGvJ3u04vdnzaYN5WoyN5LFckrlA71+CsZD1CGSqbVDWEXAMPLE  
-----END RSA PRIVATE KEY-----
```

- e. For **Certificate chain**, type 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/arxbx1BJCHBv1EPNnsdw8EXAMPLE  
-----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**.
3. After the certificate is successfully created or imported, make note of the certificate ARN. You need it when setting up the custom domain name, next.

How to Create an Edge-Optimized Custom Domain Name

Topics

- [Set Up an Edge-Optimized Custom Domain Name for an API Gateway API \(p. 436\)](#)
- [Log Custom Domain Name Creation in CloudTrail \(p. 439\)](#)
- [Configure Base Path Mapping of an API with a Custom Domain Name as its Host Name \(p. 440\)](#)
- [Rotate a Certificate Imported into ACM \(p. 441\)](#)
- [Call Your API with Custom Domain Names \(p. 441\)](#)

Set Up an Edge-Optimized Custom Domain Name for an API Gateway API

The following procedure describes how to set up a custom domain name for an API using the API Gateway console.

To set up 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 Custom Domain Name** next.
4. a. Under **New Custom Domain Name**, type your domain name (for example, `api.example.com`) in **Domain Name**.

Note

Do not use the wildcard character (i.e., *) for your custom domain names. API Gateway does not support it, even though the API Gateway console (or the AWS CLI) accepts it and can map it to a CloudFront distribution. However, you can use wildcard certificates.

- b. Choose a certificate from the **ACM Certificate** list.
- c. Choose **Add mapping** under **Base Path Mappings** to set a base path (**Path**) for a deployed API in a given stage (selected from the **Destination** dropdown lists.) You can also set the base path mapping after the custom domain name is created. For more information, see [Configure Base Path Mapping of an API with a Custom Domain Name as its Host Name \(p. 440\)](#).
- d. Choose **Save**.
5. 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.

6. In this step, we use Amazon Route 53 as an example DNS provider to show how to set up an A-record alias for your Internet domain to map the custom domain name to the associated CloudFront distribution name. The instructions can be adapted to other DNS providers.
 - a. Sign in to the Route 53 console.
 - b. Create an A-IPv4 address record set for your custom domain (e.g., api.example.com). An A-record maps a custom domain name to an IP4 address.
 - c. Choose **Yes for Alias**, type the CloudFront domain name (e.g., d3boq9ikothtgw.cloudfront.net) in **Alias Target**, and then choose **Create**. The A-record alias here maps your custom domain name to the specified CloudFront domain name that is itself mapped to an IP4 address.

Create Record Set

Name: api.haymuto.com.

Type: A – IPv4 address

Alias: Yes No

Alias Target: d3boq9ikothtgw.cloudfront.net !

Alias Hosted Zone ID: Z2FDTNDATAQYW2

You can also type the domain name for the resource. Examples:
- CloudFront distribution domain name: d111111abcdef8.cloudfront.net
- Elastic Beanstalk environment CNAME: example.elasticbeanstalk.com
- ELB load balancer DNS name: example-1.us-east-1.elb.amazonaws.com
- S3 website endpoint: s3-website.us-east-2.amazonaws.com
- Resource record set in this hosted zone: www.example.com

[Learn More](#)

Routing Policy: Simple

Route 53 responds to queries based only on the values in this record. [Learn More](#)

Evaluate Target Health: Yes No !

Create

Tip

The **Alias Hosted Zone ID** identifies the hosted zone of the specified **Alias Target**. The Route 53 console automatically fills in the value when you enter a valid domain name for **Alias Target**. To create an A-record alias without using the Route 53 console, such as when you use the AWS CLI, you must specify the required hosted zone Id. For any CloudFront

distribution domain name, the hosted zone Id value is always Z2FDTNDATAQYW2, as documented in [AWS Regions and Endpoints for CloudFront](#).

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 typed 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, as shown above. 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 Step 6 of 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-specific API Gateway account IDs of CloudFront distributions associated with a custom domain name

Region	Account ID
us-east-1	392220576650
us-east-2	718770453195
us-west-1	968246515281

Region	Account ID
us-west-2	109351309407
eu-west-1	631144002099
eu-west-2	544388816663
eu-central-1	474240146802
ap-northeast-1	969236854626
ap-northeast-2	020402002396
ap-southeast-1	195145609632
ap-southeast-2	798376113853

Configure Base Path Mapping of an API with a Custom Domain Name as its Host Name

You can use a single custom domain name as the host name 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 `yourPetStore` for the `PetShop` API. The URL of `https://api.example.com/yourPetStore` 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 \(p. 436\)](#).

To set the base path for API mappings using the API Gateway console

1. Choose a custom domain name from the list of available **Custom Domain Names** list under your account.
2. Choose **Show Base Path Mappings** or **Edit**.
3. Choose **Add mapping**.
4. (Optional) Type a base path name for **Path**, choose an API from **Destination**, and then choose a stage.

Note

The **Destination** list shows the deployed APIs under your account.

5. Choose **Save** to finish setting up the base path mapping for the API.

Note

To delete a mapping after you create it, next to the mapping that you want to delete, choose the trash icon.

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 host name. 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 \(p. 434\)](#) 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. Select the custom domain name of your choice, under the **Custom Domain Names** pane.
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	pro	<code>https://udxjef.execute-api.us-east-1.amazonaws.com/pro</code>	/petstore	<code>https://api.example.com/petstore</code>
udxjef	tst	<code>https://udxjef.execute-api.us-east-1.amazonaws.com/tst</code>	/petdepot	<code>https://api.example.com/petdepot</code>
qf3duz	dev	<code>https://qf3duz.execute-api.us-east-1.amazonaws.com/dev</code>	/bookstore	<code>https://api.example.com/bookstore</code>
qf3duz	tst	<code>https://qf3duz.execute-api.us-east-1.amazonaws.com/tst</code>	/bookstand	<code>https://api.example.com/bookstand</code>

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](#).

Set up a Custom Domain Name for a Regional API in API Gateway

As with an edge-optimized API endpoint, you can create a custom domain name for a regional API endpoint. To support a regional custom domain name, you must provide a certificate. If AWS Certificate Manager (ACM) Certificate is used, this certificate must be region-specific. If ACM is available in the region, you must provide an ACM Certificate specific to that region. If ACM is not supported in the region, you must upload a certificate to API Gateway in that region when creating the regional custom domain name. For more information about creating or uploading a custom domain name certificate, see [Get Certificates Ready in AWS Certificate Manager \(p. 434\)](#).

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 does not 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. For more information about use of the service-linked role, see [Using Service-Linked Roles](#).

When a regional custom domain name is successfully created, API Gateway returns the newly created regional custom domain name in the `domainName` property, returns its regional host name in the

`regionalDomainName` property, and returns the regional hosted zone ID in the `regionalHostedZoneId` property. You must configure your DNS records to map the regional custom domain name to its host name of the given hosted zone ID. To do so in Amazon Route 53, you must use AWS CLI or AWS SDK for Route 53. The following is an AWS CLI for Route 53 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`). The JSON file 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 host name (`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 [API Gateway Regions and Endpoints](#).

```
{
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "regional.example.com",
        "Type": "A",
        "AliasTarget": {
          "DNSName": "d-numh1z56v6.execute-api.us-west-2.amazonaws.com",
          "HostedZoneId": "Z2OJLYMU09EFXC",
          "EvaluateTargetHealth": false
        }
      }
    }
  ]
}
```

Similarly, you can run the same command to map an edge-optimized custom domain name to its associated CloudFront distribution with a different `setup-dns-record.json` file. The following example shows how to set up a DNS A-record to map an edge-optimized custom domain name (`edge.example.com`) to its CloudFront distribution name (`d1frvgze7vy1bf.cloudfront.net`) provisioned as part of the custom domain name creation.

```
{
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "edge.example.com",
        "Type": "A",
        "AliasTarget": {
          "DNSName": "d1frvgze7vy1bf.cloudfront.net",
          "HostedZoneId": "Z2FDTNDATAQYW2",
          "EvaluateTargetHealth": false
        }
      }
    }
  ]
}
```

Notice that the edge-optimized hosted zone is independent of regions and the `DNSName` takes the value of the associated CloudFront distribution name. You can also use the Route 53 management console to set up the DNS record for an edge-optimized custom domain name, but not for a regional custom domain name.

Topics

- [Set up a Regional Custom Domain Name Using the API Gateway Console \(p. 444\)](#)
- [Set up a Regional Custom Domain Name Using AWS CLI \(p. 444\)](#)
- [Set up a Regional Custom Domain Name Using the API Gateway REST API \(p. 445\)](#)

Set up a Regional Custom Domain Name 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 and choose **Custom Domain Names** in the primary navigation pane.
2. Choose **+Create New Custom Domain Name** above the **Custom Domain Names** table.
3. In **New Custom Domain Name**, type a custom domain name, for example, `my-api.example.com`, in **Domain Name**.
4. Choose **Regional** for **Endpoint Configuration**.
5. Choose a certificate from the **ACM Certificate (us-east-1)** drop-down list.
6. If you have created and deployed an API to use this custom domain name, choose **Add mapping**, type a base path under the custom domain name in **Path**, choose an API from the API drop-down list under **Destination**, and choose a stage from the **Stage** drop-down list. To add another base path mapping, repeat the step.
7. Choose **Save**.
8. Note the newly provisioned target domain name and then go to your DNS provider. Create a DNS record to point the newly created regional domain name to this target domain name.

Set up a Regional Custom Domain Name 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 of the `REGIONAL` type and the ARN of a regional certificate.

```
aws apigateway create-domain-name \
    --domain-name 'regional.example.com' \
    --endpoint-configuration types=REGIONAL \
    --regional-certificate-arn 'arn:aws:acm:us-west-2:123456789012:certificate/
c19332f0-3be6-457f-a244-e03a423084e6'
```

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:

```
{  
    "certificateUploadDate": "2017-10-13T23:02:54Z",  
    "domainName": "regional.example.com",
```

```

    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "regionalCertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/
c19332f0-3be6-457f-a244-e03a423084e6",
    "regionalDomainName": "d-numh1z56v6.execute-api.us-west-2.amazonaws.com"
}

```

The `regionalDomainName` property value returns the regional API's host name. 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 host name.

If you set the endpoint type to `EDGE` or do not set the type at all, you create an edge-optimized custom domain name. The output contains the `distributionDomainName` instead of `regionalDomainName`. The `distributionName` property value returns the API's edge-optimized host name. You must create a DNS record to point the custom domain name to this distribution domain name. This enables the traffic that is bound to the custom domain name to be routed to the API's edge-optimized host name.

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 host name.
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 apigateway create-base-path-mapping \
--domain-name 'regional.example.com' \
--base-path 'RegionalApiTest' \
--rest-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/RegionalApiTest`.

Set up a Regional Custom Domain Name Using the API Gateway REST API

To create a custom domain name for a regional API using the API Gateway REST API

1. Follow the [domainname:create](#) link-relation to create a custom domain name of the `REGIONAL` endpoint type, specifying the regional certificate by using its ARN.

```

POST /domainnames HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c

{
    "domainName": "regional.example.com",
    "regionalCertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/
c19332f0-3be6-457f-a244-e03a423084e6",
    "endpointConfiguration" : {
        "types" : ["REGIONAL"]
    }
}

```

}

Note that to set up a regional custom domain name, you set the required certificate ARN on the input property of `regionalCertificateArn`. In contrast, to create an edge-optimized custom domain name, you set the required certificate ARN on the input property of `certificateArn`.

The successful response has a 201 `Created` status code and a payload similar to the following:

```
{
    "_links": {
        ...
    },
    "certificateUploadDate": "2017-10-13T23:02:54Z",
    "domainName": "regional.example.com",
    "endpointConfiguration": {
        "types": "REGIONAL"
    },
    "regionalCertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/c19332f0-3be6-457f-a244-e03a423084e6",
    "regionalDomainName": "d-numh1z56v6.execute-api.us-west-2.amazonaws.com."
}
```

For the given custom domain name (for example, `regional.example.com`), API Gateway returns the associated regional domain name (for example, `d-numh1z56v6.execute-api.us-west-2.amazonaws.com`) as the API's regional host name. You must create a DNS record to point the custom domain name to this regional domain name. This enables the traffic that is bound to the custom domain name to be routed to the API's regional host name. The DNS record can be of the CNAME or A type.

If you set the endpoint configuration type to `EDGE` or do not set the type at all, you create an edge-optimized custom domain name. The output contains the `distributionDomainName` instead of `regionalDomainName`. The `distributionDomainName` value shows the API's edge-optimized host name. You must create a DNS record to point the custom domain name to this distribution domain name. This enables the traffic that is bound to the custom domain name to be routed to the API's edge-optimized host name.

2. Set up DNS records in your DNS provider to point the custom domain name to the regional API host name. This enables traffic that is bound to the custom domain name to be routed to the regional API host name. In Route 53, you can set the CNAME or Alias A record using the AWS CLI, an AWD SDK, or the Route 53 REST API.
3. With the new custom domain name created, you set a base path on the domain name to target one of the regional APIs. Assuming you deployed a regional API (`0qzs2sy7bh`) to a test stage, you can add this API to the domain name's base path mappings by calling `basepathmapping:create` from the API Gateway REST API:

```
POST /domainnames/regional.example.com/basepathmappings HTTP/1.1Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date, Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c

{
    "basePath" : "testRegionalApi",
    "restApiId" : "0qzs2sy7bh",
    "stage" : "test"
}
```

With the base path mapping set, you can now call the API by using its custom domain name. With the regional PetStore example API, use the following REST API request to call `GET /pets`:

```
https://regional.example.com/testRegionalApi/pets
```

To call `GET /pets/{petId}`, make the following API request:

```
https://regional.example.com/testRegionalApi/pets/1
```

Migrate 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 Regional and Edge-Optimized Domain Names Using the API Gateway Console \(p. 447\)](#)
- [Update Custom Domain Names Using the AWS CLI \(p. 448\)](#)
- [Update Custom Domain Names Using the API Gateway REST API \(p. 450\)](#)

Migrate Regional and Edge-Optimized Domain Names Using the API Gateway Console

To use the API Gateway console to migrate a regional custom domain name to an edge-optimized custom domain name and vice versa, use the following procedure.

To migrate a regional or edge-optimized custom domain name using the API Gateway console

1. Sign in to the API Gateway console and choose **Custom Domain Names** in the primary navigation pane.
2. Choose an existing domain name from **Custom Domain Names**, and then choose **Edit**.
3. Depending on the existing endpoint type, do the following:
 - a. For an edge-optimized domain name, choose **Add Regional Configuration**.
 - b. For a regional domain name, choose **Add Edge Configuration**.

4. Choose a certificate from the drop-down list.
5. Choose **Save**.
6. Choose **Proceed** to confirm adding the new endpoint.
7. Update the DNS records to point the new domain name to the newly provisioned target domain name.

Update Custom Domain Names Using the AWS CLI

To use the AWS CLI to update 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

- [Update an Edge-Optimized Custom Domain Name to Regional \(p. 448\)](#)
- [Update a Regional Custom Domain Name to Edge-Optimized \(p. 449\)](#)

Update 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](#) command of AWS CLI, 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 updated regional custom domain name, the resulting `regionalDomainName` property returns the regional API host name. You must set up a DNS record to point the regional custom domain name to this regional host name. 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'} \
]
```

Update 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 host name 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 host name.

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"  
    }  
}
```

Update Custom Domain Names Using the API Gateway REST API

To use the API Gateway REST API to update an edge-optimized custom domain name to a regional one, or from a regional custom domain name to an edge-optimized one, use the [domainname:update](#) link-relation and, optionally, the [domainname:delete](#) link-relation.

Topics

- [Update an Edge-Optimized Custom Domain Name to Regional \(p. 450\)](#)
- [Update a Region Domain Name to Edge-Optimized \(p. 451\)](#)

Update an Edge-Optimized Custom Domain Name to Regional

To update an edge-optimized custom domain name (`api.example.com`) to a regional custom domain name, call [domainname:update](#) of the API Gateway REST API:

```
PATCH /domainnames/api.example.com HTTP/1.1  
Host: apigateway.us-west-2.amazonaws.com  
Content-Type: application/x-amz-json-1.0  
X-Amz-Date: 20170511T214723Z  
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/  
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,  
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c  
  
{  
    "patchOperations": [  
        {  
            "op" : "add",  
            "path" : "/endpointConfiguration/types",  
            "value" : "REGIONAL"  
        },  
        {  
            "op" : "add",  
            "path" : "/regionalCertificateArn",  
            "value" : "arn:aws:acm:us-west-2:123456789012:certificate/c19332f1-3be7-457f-a245-  
e03a423084e7"  
        }  
    ]  
}
```

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/c19332f1-3be7-457f-a245-e03a423084e7",
    "regionalDomainName": "d-gdisjchyn5.execute-api.us-west-2.amazonaws.com"
}
```

For the regional custom domain name, the returned `regionalDomainName` property value is the regional API host name. You must set up a DNS record to point the regional custom domain name to this regional API host name. This enables traffic that is bound to the regional custom domain name to be routed to the regional API host.

You can then remove the edge-optimized API custom domain name:

```
PATCH /domainnames/{domain-name} HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c

{
    "patchOperations": [
        {
            "op" : "remove",
            "path" : "/endpointConfiguration/types"
        },
        {
            "op" : "remove",
            "path" : "/certificateName"
        },
        {
            "op" : "remove",
            "path" : "/certificateArn"
        }
    ]
}
```

Update a Region Domain Name to Edge-Optimized

To migrate a regional custom domain name to an edge-optimized custom domain name, call `domainname:update` from the API Gateway REST API, as follows:

```
PATCH /domainnames/{domain-name} HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c
```

```
{
    "patchOperations" : [ {
        "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"
    } ]
}
```

For an edge-optimized API custom domain name, the ACM Certificate must be from the `us-east-1` region.

The successful 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 domain name of an Amazon CloudFront distribution. You must set a DNS record to point the custom domain name to this distribution domain name, so that traffic to the custom domain name is routed to the named CloudFront distribution.

You can then remove the regional API custom domain name:

```
PATCH /domainnames/{domain-name} HTTP/1.1
Host: apigateway.us-west-2.amazonaws.com
Content-Type: application/x-amz-json-1.0
X-Amz-Date: 20170511T214723Z
Authorization: AWS4-HMAC-SHA256 Credential={ACCESS-KEY-ID}/20170511/us-west-2/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
Signature=d0abd98a2a06199531c2916b162ede9f63a247032cdc8e4d077216446d13103c

{
    "patchOperations": [
        {
            "op" : "remove",
            "path" : "/endpointConfiguration/types"
        },
        {

```

```
        "op" : "remove",
        "path" : "/regionalCertificateArn"
    }
}
```

The successful response to the preceding request 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": "regional.haymuto.com",
    "endpointConfiguration": {
        "types": "EDGE"
    }
}
```

Sell Your API Gateway API through AWS Marketplace

After you build, test, and deploy your API, you can package it 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 \(p. 314\)](#).

To sell your API 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.

To enable your customers to buy your product on AWS Marketplace, you must register your developer portal (an external application) with AWS Marketplace. The developer portal must handle the subscription requests that are redirected from the AWS Marketplace console.

For more information about selling your API as a SaaS product on AWS Marketplace, see [AWS Marketplace SaaS Subscriptions - Seller Integration Guide](#).

Topics

- [Initialize AWS Marketplace Integration with API Gateway \(p. 453\)](#)
- [Handle Customer Subscription to Usage Plans \(p. 455\)](#)

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 API 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 API.

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` header parameter containing buyer information. Follow the instructions in the Register Application section of the [SaaS Seller Integration Guide](#) 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 Step 6.4 of the [SaaS Seller Integration Guide](#)). 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 API.

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"
        }
}
```

Invoking an API in Amazon API Gateway

Calling a deployed API involves submitting requests to the API Gateway component service for API execution, known as `execute-api`. The root URL of such requests is of the following format:

```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

where `{restapi_id}` is the API identifier, `{region}` is the API deployed region, and `{stage_name}` is the stage name of an API deployment.

You can find this root URL in the given **Stage Editor**. It is listed as the **Invoke URL** at the top. If the API's root resource exposes a GET method without requiring user authentication, you can call the method by clicking the **Invoke URL** link. You can also construct this root URL by combining the `host` and `basePath` fields of an exported Swagger definition file of the API.

If an API permits anonymous access, you can use any web browser to invoke any GET-method calls by copying and pasting an appropriate invocation URL to the browser's address bar. For other methods or any authentication-required calls, the invocation will be more involved because you must specify a payload or sign the requests. You can handle these in a script behind an HTML page or in a client app using one of the AWS SDKs.

For testing, you can use the API Gateway console to call an API using the API Gateway's `TestInvoke` feature, which bypasses the `Invoke URL` and allows API testing before the API is deployed. Alternatively, you can use the [Postman](#) app to test a successfully deployed API, without writing a script or a client.

Note

Query string parameter values in an invocation URL cannot contain %%.

Topics

- [Obtain an API's Invoke URL in the API Gateway Console \(p. 457\)](#)
- [Use the API Gateway Console to Test a Method \(p. 458\)](#)
- [Use Postman to Call an API \(p. 459\)](#)
- [Call API through Generated SDKs \(p. 459\)](#)
- [Call API through AWS Amplify JavaScript Library \(p. 477\)](#)
- [Trace API Management and Invocation \(p. 477\)](#)

Important

You must have already deployed the API in API Gateway. Follow the instructions in [Deploying an API in Amazon API Gateway \(p. 370\)](#).

Obtain an API's Invoke URL in the API Gateway Console

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API you want to call, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.

4. The URL displayed next to **Invoke URL** should look something like this, where *my-api-id* is the identifier API Gateway assigns to your API, *region-id* is the AWS region identifier (for example, `us-east-1`) where you deployed your API, and *stage-name* is the name of the stage for the API you want to call:

```
https://my-api-id.execute-api.region-id.amazonaws.com/stage-name/{resourcePath}
```

Depending on the method type you want to call and the tool you want to use, copy this URL to your clipboard, and then paste and modify it to call the API from a web browser, a web debugging proxy tool or the cURL command-line tool, or from your own API.

If you are not familiar with which method to call or the format you must use to call it, browse the list of available methods by following the instructions in [View a Methods List in API Gateway \(p. 237\)](#).

To call the method directly from the API Gateway console, see [Use the Console to Test a Method \(p. 458\)](#).

For more options, contact the API owner.

Use the API Gateway Console to Test a Method

Use the API Gateway console to test a method.

Topics

- [Prerequisites \(p. 458\)](#)
- [Test a Method with the API Gateway Console \(p. 458\)](#)

Prerequisites

- You must specify the settings for the methods you want to test. Follow the instructions in [Set up API Methods in API Gateway \(p. 106\)](#).

Test a Method with the API Gateway Console

Important

Testing methods with the API Gateway console may 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.

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. In the box that contains the name of the API for the method, choose **Resources**.
3. In the **Resources** pane, choose the method you want to test.
4. In the **Method Execution** pane, in the **Client** box, choose **TEST**. Type values in any of the displayed boxes (such as **Query Strings**, **Headers**, and **Request Body**).

For additional options you may 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 may be different from those sent from the Lambda function, HTTP proxy, or AWS service proxy.

- **Logs** are 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](#).

Use Postman to Call an API

Use the [Postman](#) app is a convenient tool to test an API in API Gateway. The following instructions walk you through the essential steps of using the Postman app to call an API. For more information, see the Postman [help](#).

1. Launch Postman.
2. Enter the endpoint URL of a request in the address bar and choose the appropriate HTTP method from the drop-down list to the left of the address bar.
3. If required, choose the **Authorization** tab. Choose **AWS Signature** for the authorization **Type**. Enter your AWS IAM user's access key ID in the **AccessKey** input field. Enter your IAM user secret key in **SecretKey**. Specify an appropriate AWS region that matches the region specified in the invocation URL. Enter **execute-api** in **Service Name**.
4. Choose the **Headers** tab. Optionally, delete any existing headers. This can clear any stale settings that may cause errors. Add any required custom headers. For example, if API keys are enabled, you can set the **x-api-key: {api_key}** name/value pair here.
5. Choose **Send** to submit the request and receive a response.

For an example of using Postman, see [Call an API with API Gateway Lambda Authorizers \(p. 282\)](#).

Call 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 \(p. 460\)](#)
- [Use an Android SDK Generated by API Gateway \(p. 463\)](#)
- [Use a JavaScript SDK Generated by API Gateway \(p. 465\)](#)
- [Use a Ruby SDK Generated by API Gateway \(p. 466\)](#)
- [Use iOS SDK Generated by API Gateway in Objective-C or Swift \(p. 469\)](#)

Use a Java SDK Generated by API Gateway

In this section, we outline the steps to use a Java SDK generated by API Gateway, by using the [Simple Calculator \(p. 427\)](#) API as an example. Before proceeding, you must complete the steps in [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#).

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 the [JDK](#) (must be version 1.8 or later).
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 to create a client project stub to call the API using the installed SDK library.

```
mvn -B archetype:generate \
    -DarchetypeGroupId=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 `DarchetypeGroupId` 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

In this section, we will outline the steps to use an Android SDK generated by API Gateway of an API. Before proceeding further, you must have already completed the steps in [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#).

Note

The generated SDK is not compatible with Android 4.4 and earlier. For more information, see [Known Issues \(p. 585\)](#).

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 the [JDK](#) (preferably version 1.7 or later).
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

Note

These instructions assume you have already completed the instructions in [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#).

To install, initiate and call a JavaScript SDK generated by API Gateway

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 [Enable CORS for a Resource \(p. 267\)](#).
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

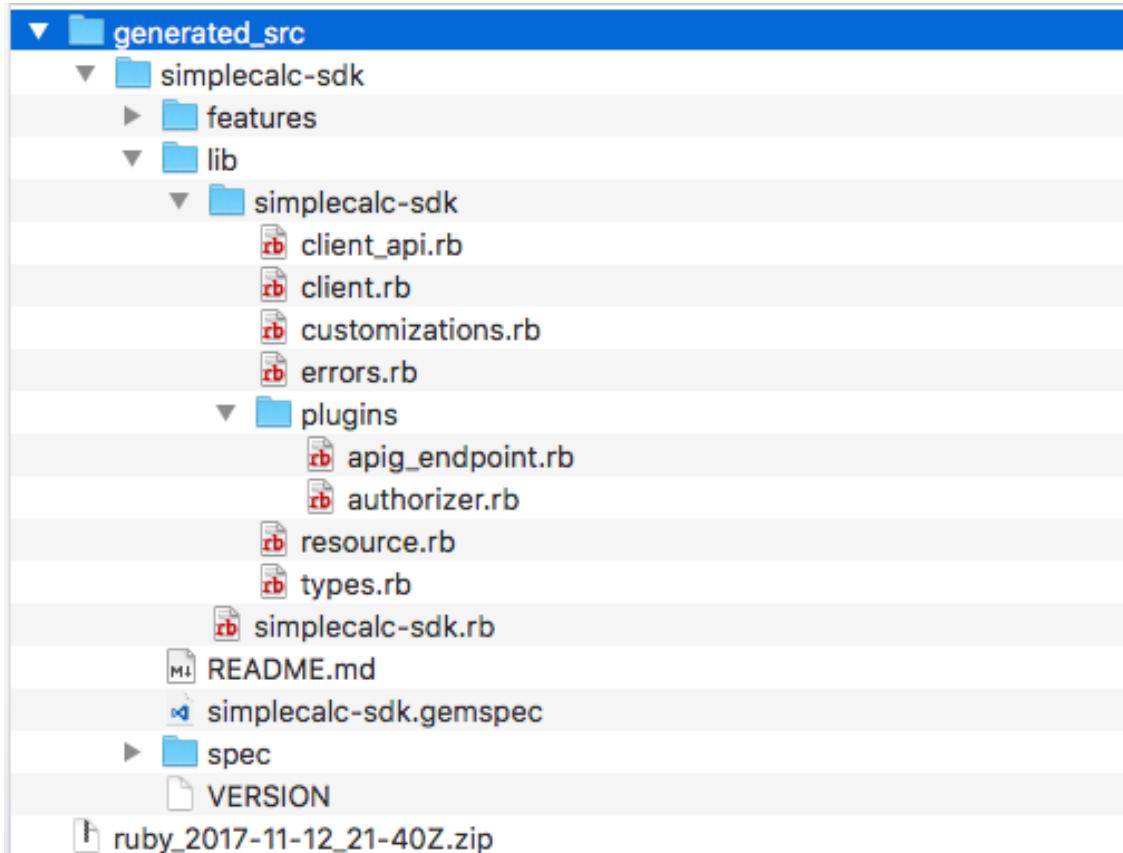
Use a Ruby SDK Generated by API Gateway

Note

These instructions assume you already completed the instructions in [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#).

To install, instantiate, and call a Ruby SDK generated by API Gateway

1. Unzip the downloaded Ruby SDK file. The generated SDK source is shown as follows.



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: 'ACCESS_KEY',
  secret_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 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 in Objective-C or Swift

In this tutorial, we will show how to use an iOS SDK generated by API Gateway in an Objective-C or Swift app to call the underlying API. We will use the [SimpleCalc API \(p. 422\)](#) 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 \(p. 469\)](#)
- [Use Generated iOS SDK \(Swift\) to Call API \(p. 473\)](#)

Use Generated iOS SDK (Objective-C) to Call API

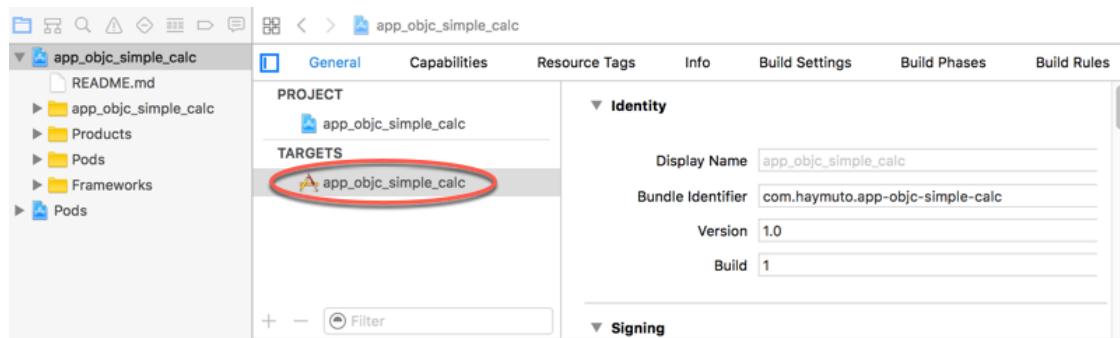
Before beginning the following procedure, you must complete the steps in [Generate SDKs for an API Using the API Gateway Console \(p. 416\)](#) for iOS in Objective-C and download the .zip file of the generated SDK.

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 \(p. 422\)](#), 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**.



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:

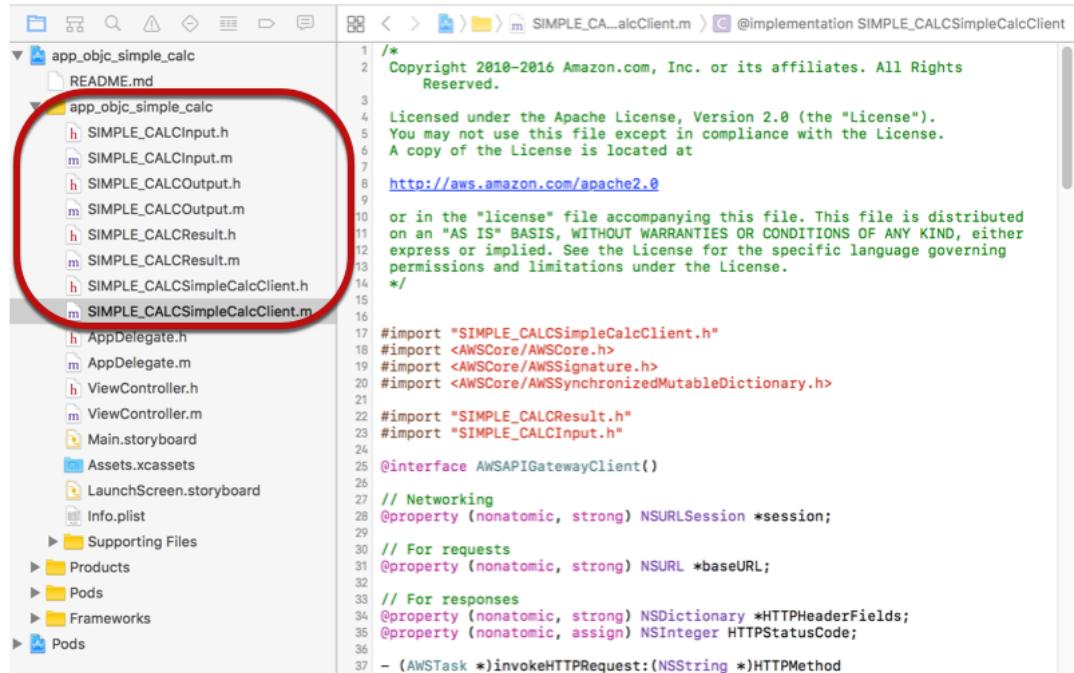
```
pod 'AWSAPIGateway', '~> 2.4.7'
```

- 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.



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 \(p. 422\)](#) 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 \(p. 420\)](#) 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 \(p. 416\)](#) for iOS in Swift and download the .zip file of the generated SDK.

Topics

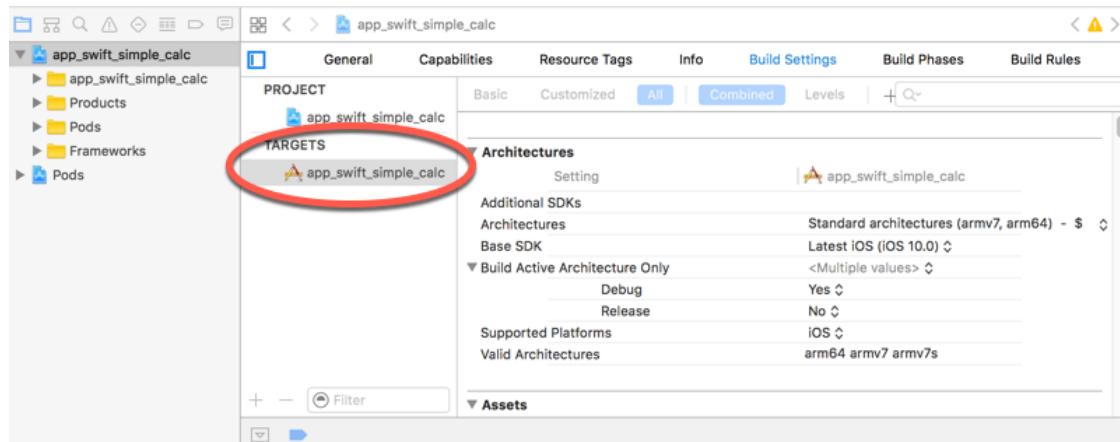
- [Install AWS Mobile SDK and API Gateway-Generated SDK in a Swift Project \(p. 473\)](#)
- [Call API methods through the iOS SDK generated by API Gateway in a Swift Project \(p. 475\)](#)

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 \(p. 422\)](#), 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.



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:

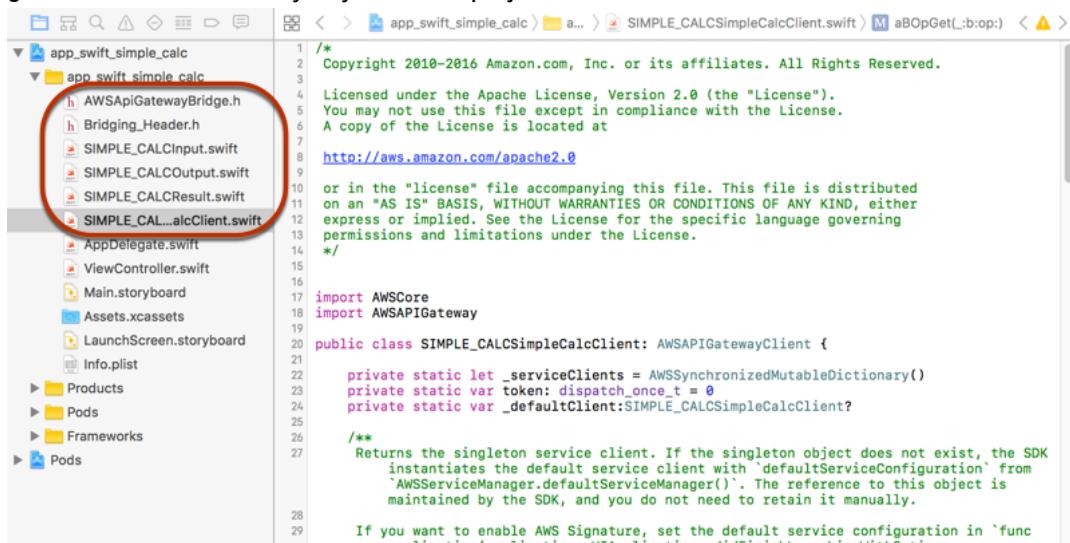
```
pod 'AWSAPIGateway', '~> 2.4.7'
```

- c. Open a terminal window and run the following command in the app directory:

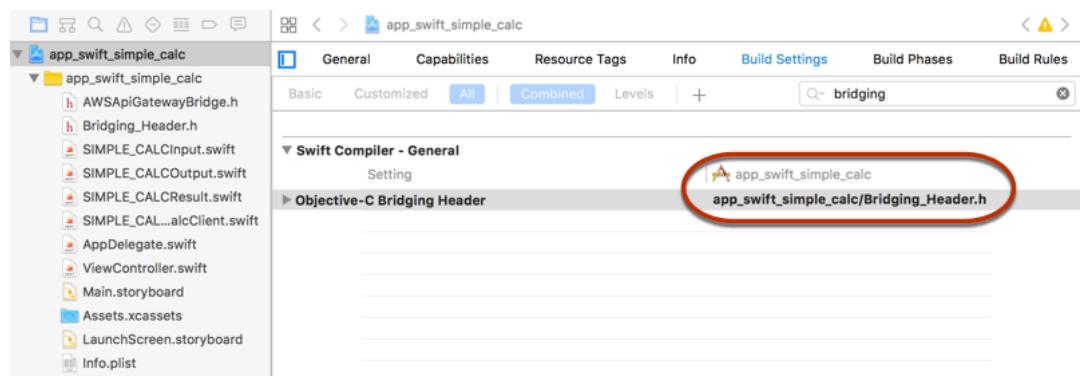
```
pod install
```

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.



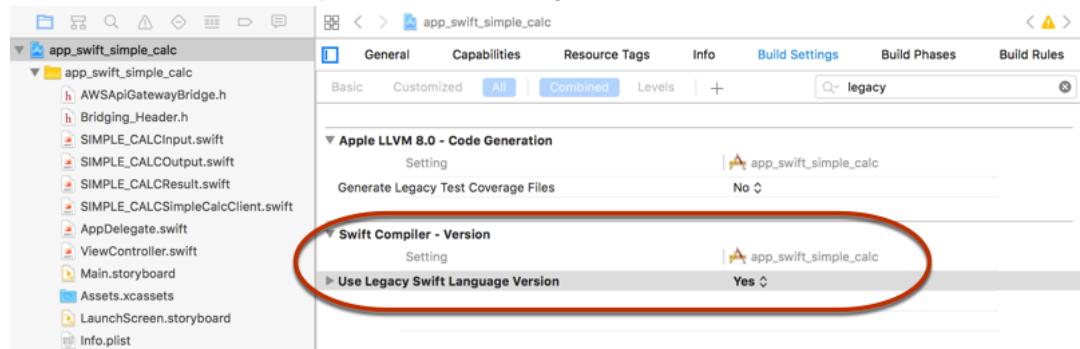
- f. To enable calling the Objective-C libraries of the AWS Mobile SDK from your Swift code project, set the `Bridging_Header.h` file path on the **Objective-C Bridging Header** property under the **Swift Compiler - General** setting of your Xcode project configuration:



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:



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 \(p. 422\)](#) 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_CALCClient.defaultClient()
```

To use Amazon Cognito with the API, set a default AWS service configuration (shown following) before getting the `defaultClient` 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 $1 \text{ div } 2 = 0.5$. Here, div is used in place of / because the [simple Lambda function \(p. 420\)](#) in the backend does not handle URL encoded path variables.

Call API through AWS Amplify JavaScript Library

The AWS Amplify JavaScript Library can be used for making API requests to API Gateway. For more information, see the instructions in the [AWS Amplify API Guide](#).

Trace API Management and Invocation

Topics

- [Log API management calls to Amazon API Gateway Using AWS CloudTrail \(p. 478\)](#)
- [Monitor API execution with Amazon CloudWatch \(p. 479\)](#)

For API execution, API Gateway automatically reports to Amazon CloudWatch your API's execution metrics on the API- and stage-levels, provided that your account has an IAM role with permissions to write logs into CloudWatch configured.

You can also opt in for API Gateway to send to CloudWatch method-level metrics using the [API Gateway console \(p. 374\)](#) or calling the [API Gateway REST API](#) or one of its SDKs. Based on these metrics, you can set CloudWatch custom alarms for troubleshooting any performance issues of your APIs.

The CloudWatch metrics include statistics about caching, latency and detected errors. You can inspect the CloudWatch logs to troubleshoot your API implementation or execution using the API dashboard in the API Gateway console or using the CloudWatch console. For more information about CloudWatch, see the [Amazon CloudWatch User Guide](#).

For API management operations, you can create AWS CloudTrail trails to log events involved in the API Gateway REST API calls. You can create the trails in the CloudTrail console.

You can use the logs to troubleshoot API creation, deployment and updates. You can also use Amazon CloudWatch to monitor the CloudTrail logs. To learn more about CloudTrail, see the [AWS CloudTrail User Guide](#).

Note

CloudTrail logs API Gateway REST API calls an API developer or owner made against the `apigateway` component, whereas CloudWatch logs API calls an API customer or client made against the `execute-api` component of API Gateway.

Log API management calls to Amazon API Gateway Using AWS CloudTrail

You can use AWS CloudTrail to capture API Gateway REST API calls in your AWS account and deliver the log files to an Amazon S3 bucket you specify. Examples of these API calls include creating a new API, resource, or method in API Gateway. CloudTrail captures such API calls from the API Gateway console or from the API Gateway APIs directly. Using the information collected by CloudTrail, you can determine which request was made to API Gateway, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

API Gateway Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to API Gateway actions are tracked in log files. API Gateway records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All of the API Gateway actions are logged and documented in the [API Gateway REST API \(p. 581\)](#). For example, calls to create a new API, resource, or method in API Gateway generate entries in CloudTrail log files.

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the `userIdentity` field in the [CloudTrail Event Reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered so you can take action quickly. For more information, see [Configuring Amazon SNS Notifications](#).

You can also aggregate API Gateway log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see [Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket](#).

Understanding API Gateway Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry that demonstrates the API Gateway get resource 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 ...  
    ]  
}
```

Monitor API execution with Amazon CloudWatch

You can monitor API execution 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 two weeks, so that you can access historical information and gain a better perspective on how your web application or service is performing. By default, API Gateway metric data is automatically sent to CloudWatch in one-minute periods. For more information, see [What Are Amazon CloudWatch](#), [Amazon CloudWatch Events](#), and [Amazon CloudWatch Logs?](#) in the [Amazon CloudWatch User Guide](#).

The metrics reported by API Gateway provide information that you can analyze in different ways. The list below shows some common uses for the metrics. These are suggestions to get you started, not a comprehensive list.

- 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 \(p. 480\)](#)
- [View CloudWatch Metrics with the API Dashboard in API Gateway \(p. 482\)](#)
- [View API Gateway Metrics in the CloudWatch Console \(p. 483\)](#)
- [View API Gateway Log Events in the CloudWatch Console \(p. 483\)](#)
- [Monitoring Tools in AWS \(p. 484\)](#)

Amazon API Gateway Dimensions and Metrics

The metrics and dimensions that API Gateway sends to Amazon CloudWatch are listed below. For more information, see [Monitor API execution with Amazon CloudWatch \(p. 479\)](#).

API Gateway Metrics

Amazon API Gateway sends metric data to CloudWatch every minute.

The `AWS/ApiGateway` namespace includes the following metrics.

Metric	Description
4XXError	<p>The number of client-side errors captured in a specified period.</p> <p>The <code>Sum</code> statistic represents this metric, namely, the total count of the 4XXError errors in the given period. The <code>Average</code> 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 <code>Count</code> metric (below).</p> <p>Unit: Count</p>
5XXError	<p>The number of server-side errors captured in a given period.</p> <p>The <code>Sum</code> statistic represents this metric, namely, the total count of the 5XXError errors in the given period. The <code>Average</code> 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 <code>Count</code> metric (below).</p> <p>Unit: Count</p>

Metric	Description
CacheHitCount	<p>The number of requests served from the API cache in a given period.</p> <p>The Sum statistic represents this metric, namely, the total count of the cache hits in the specified 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 period. The denominator corresponds to the Count metric (below).</p> <p>Unit: Count</p>
CacheMissCount	<p>The number of requests served from the back end in a given period, when API caching is enabled.</p> <p>The Sum statistic represents this metric, namely, the total count of the cache misses in the specified period. The Average statistic represents the cache miss rate, namely, the total count of the cache hits divided by the total number of requests during the period. The denominator corresponds to the Count metric (below).</p> <p>Unit: Count</p>
Count	<p>The total number API requests in a given period.</p> <p>The SampleCount statistic represents this metric.</p> <p>Unit: Count</p>
IntegrationLatency	<p>The time between when API Gateway relays a request to the back end and when it receives a response from the back end.</p> <p>Unit: Millisecond</p>
Latency	<p>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.</p> <p>Unit: Millisecond</p>

Dimensions for Metrics

You can use the dimensions in the following table to filter API Gateway metrics.

Dimension	Description
ApiName	Filters API Gateway metrics for an API of the specified API name.
ApiName, Method, Resource, Stage	Filters API Gateway metrics for an API method of the specified API, stage, resource, and method.

Dimension	Description
	<p>API Gateway will not send such metrics unless you have explicitly enabled detailed CloudWatch metrics. You can do this in the console by selecting Enable CloudWatch Metrics under a stage Settings tab. Alternatively, you can call the stage:update action of the API Gateway REST API to update the <code>metricsEnabled</code> property to <code>true</code>.</p> <p>Enabling such metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing.</p>
ApiName, Stage	Filters API Gateway metrics for an API stage of the specified API 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 \(p. 482\)](#)
- [Examine API activities in the Dashboard \(p. 482\)](#)

Prerequisites

1. You must have an API created in API Gateway. Follow the instructions in [Creating an API in Amazon API Gateway \(p. 81\)](#).
2. You must have the API deployed at least once. Follow the instructions in [Deploying an API in Amazon API Gateway \(p. 370\)](#).
3. To get CloudWatch metrics for individual methods, you must have CloudWatch Logs enabled for those methods in a given stage. The process is prescribed in [Update Stage Settings \(p. 374\)](#). Your account will be charged for accessing method-level logs, but not for accessing API- or stage-level logs.

Examine API activities in the Dashboard

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. Choose the name of the API.
3. Under the selected API, choose **Dashboard**.
4. To display a summary of API activity over time, for **Stage**, choose the desired stage.
5. Use **From** and **To** to enter the date range.
6. Refresh, if needed, and view individual metrics displayed in separate graphs titled **API Calls**, **Integration Latency**, **Latency**, **4xx Error** and **5xx Error**. The **CacheHitCount** and **CacheMissCount** graphs will be displayed only if API caching has been enabled.

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 \(p. 374\)](#).

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 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

To view logged API requests and responses using the CloudWatch console

1. In the navigation pane, choose **Logs**.
2. Under the **Log Groups** table, choose a log group of the **API-Gateway-Execution-Logs_{rest-api-id}/ {stage-name}** name.
3. Under the **Log Streams** table, choose a log stream. You can use the timestamp to help locate the log stream of your interest.
4. Choose **Text** to view raw text or choose **Row** to view the event row by row.

Note

CloudWatch lets you delete log groups or streams. However, you should refrain from deleting API Gateway API log groups or streams and let API Gateway manage these resources. Manually deleting log groups or streams may cause API requests and responses not 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.

Also failed requests due to throttling (429) or access (403) errors are not logged and will not be included in the report.

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 [Monitor API execution with Amazon CloudWatch \(p. 479\)](#).
- **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.

API Gateway Extensions to Swagger

The API Gateway extensions support the AWS-specific authorization and API Gateway-specific API integrations. In this section, we will describe the API Gateway extensions to the Swagger specification.

Tip

To understand how the API Gateway extensions are used in an app, you can use the API Gateway console to create an API and export it to a Swagger definition file. For more information on how to export an API, see [Export an API \(p. 414\)](#).

Topics

- [x-amazon-apigateway-any-method Object \(p. 486\)](#)
- [x-amazon-apigateway-api-key-source Property \(p. 487\)](#)
- [x-amazon-apigateway-authorizer Object \(p. 488\)](#)
- [x-amazon-apigateway-authtype Property \(p. 491\)](#)
- [x-amazon-apigateway-binary-media-types Property \(p. 492\)](#)
- [x-amazon-apigateway-documentation Object \(p. 492\)](#)
- [x-amazon-apigateway-gateway-responses Object \(p. 493\)](#)
- [x-amazon-apigateway-gateway-responses.gatewayResponse Object \(p. 493\)](#)
- [x-amazon-apigateway-gateway-responses.responseParameters Object \(p. 494\)](#)
- [x-amazon-apigateway-gateway-responses.responseTemplates Object \(p. 495\)](#)
- [x-amazon-apigateway-integration Object \(p. 496\)](#)
- [x-amazon-apigateway-integration.requestTemplates Object \(p. 499\)](#)
- [x-amazon-apigateway-integration.requestParameters Object \(p. 500\)](#)
- [x-amazon-apigateway-integration.responses Object \(p. 501\)](#)
- [x-amazon-apigateway-integration.response Object \(p. 502\)](#)
- [x-amazon-apigateway-integration.responseTemplates Object \(p. 503\)](#)
- [x-amazon-apigateway-integration.responseParameters Object \(p. 504\)](#)
- [x-amazon-apigateway-request-validator Property \(p. 504\)](#)
- [x-amazon-apigateway-requestValidators Object \(p. 505\)](#)
- [x-amazon-apigateway-requestValidators.requestValidator Object \(p. 506\)](#)

x-amazon-apigateway-any-method Object

Specifies the [Swagger Operation Object](#) for the API Gateway catch-all ANY method in a [Swagger Path Item Object](#). This object can exist alongside other Operation objects and will catch any HTTP method that was not explicitly declared.

The following table lists the properties extended by API Gateway. For the other Swagger Operation properties, see the Swagger specification.

Properties

Property Name	Type	Description
x-amazon-apigateway-integration	x-amazon-apigateway-integration Object (p. 496)	Specifies the integration of the method with the backend.

Property Name	Type	Description
		This is an extended property of the Swagger Operation object. The integration can be of type AWS, AWS_PROXY, HTTP, HTTP_PROXY, or MOCK.

x-amazon-apigateway-any-method Example

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",
      "passthroughBehavior": "when_no_match",
      "httpMethod": "POST",
      "type": "aws_proxy"
    }
  }
}
```

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.

```
{
  "swagger" : "2.0",
  "info" : {
    "title" : "Test1"
```

```

    },
    "schemes" : [ "https" ],
    "basePath" : "/import",
    "x-amazon-apigateway-api-key-source" : "HEADER",
    .
    .
}
```

x-amazon-apigateway-authorizer Object

Defines a Lambda authorizer (formerly known as a custom authorizer) to be applied for authorization of method invocations in API Gateway. This object is an extended property of the [Swagger Security Definitions object](#).

Properties

Property Name	Type	Description
type	string	The type of the authorizer. This is a required property and the value must be "token", for an authorizer with the caller identity embedded in an authorization token, or "request", for an authorizer with the caller identity contained in request parameters.
authorizerUri	string	The Uniform Resource Identifier (URI) of the authorizer Lambda function. The syntax is as follows: <div style="border: 1px solid black; padding: 5px; width: fit-content;"> "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" </div>
authorizerCredentials	string	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 ".
identitySource	string	Comma-separated list of mapping expressions of the request parameters as the identity source. Applicable for the authorizer of the "request" type only.
identityValidationExpression	string	A regular expression for validating the token as the

Property Name	Type	Description
		incoming identity. For example, " <code>^x-[a-z]+</code> ".
<code>authorizerResultTtlInSeconds</code>	<code>string</code>	The number of seconds during which the resulting IAM policy is cached.

x-amazon-apigateway-authorizer Examples

The following Swagger 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 Swagger operation object snippet sets the `GET /http` to use the Lambda authorizer specified above.

```

"/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 Swagger 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 Swagger 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 Swagger 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
}
}
```

x-amazon-apigateway-authtype Property

Specify an optional customer-defined information describing a Lambda authorizer (formerly known as a custom authorizer). It is used for API Gateway API import and export without functional impact.

This property is an extended property of the [Swagger Security Definitions Operation](#) object.

x-amazon-apigateway-authtype Example

The following example sets the type of a Lambda authorizer using OAuth 2.

```
"cust-authorizer" : {
    "type" : "...", // required
    "name" : "...", // name of the identity source header
    "in" : "header", // must be header
    "x-amazon-apigateway-auth-type" : "oauth2", // Specifies the authorization mechanism
for the client.
    "x-amazon-apigateway-authorizer" : {
        ...
    }
}
```

The following security definition example specifies authorization using AWS Signature Version 4:

```
"sigv4" : {  
    "type" : "apiKey",  
    "name" : "Authorization",  
    "in" : "header",  
    "x-amazon-apigateway-authtype" : "awsSigv4"  
}
```

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`, `image/jpeg`, etc. This extension is a JSON Array.

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
<code>documentationParts</code>	Array	An array of the exported or imported <code>DocumentationPart</code> instances.
<code>version</code>	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 Swagger 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-gateway-responses Object

Defines the gateway responses for an API as a string-to-[GatewayResponse](#) map of key-value pairs.

Properties

Property Name	Type	Description
<code>responseType</code>	x-amazon-apigateway-gateway-responses.gatewayResponse (p. 493)	A GatewayResponse for the specified <code>responseType</code> .

x-amazon-apigateway-gateway-responses Example

The following API Gateway extension to Swagger example defines a [GatewayResponses](#) map containing 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
<code>responseParameters</code>	x-amazon-apigateway-gateway-responses.responseParameters (p. 494)	Specifies the GatewayResponse parameters, namely the header

Property Name	Type	Description
		parameters. The parameter values can take any incoming request parameter (p. 187) value or a static custom value.
<i>responseTemplates</i>	x-amazon-apigateway-gateway- responses.responseTemplates (p. 456)	Specifies the mapping templates of the gateway response. The templates are not processed by the VTL engine.
<i>statusCode</i>	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 Swagger 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 "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.

Properties

Property Name	Type	Description
<i>gatewayresponse.param-position.param-name</i>	string	<i>param-position</i> can be <code>header</code> , <code>path</code> or <code>querystring</code> . For more information, see Map Method Request Data to Integration Request Parameters (p. 187) .

x-amazon-apigateway-gateway- responses.responseParameters Example

The following Swagger 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 is not processed by the [Velocity Template Language \(VTL\)](#) engine.

Properties

Property Name	Type	Description
<code>content-type</code>	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 Swagger 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 Swagger 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-integration Object

Specifies details of the backend integration used for this method. This extension is an extended property of the [Swagger Operation](#) object. The result is an [API Gateway integration](#) object.

Properties

Property Name	Type	Description
cacheKeyParameters	An array of <code>string</code>	A list of request parameters whose values are to be cached.
cacheNamespace	<code>string</code>	An API-specific tag group of related cached parameters.
connectionId	<code>string</code>	The ID of a VpcLink for the private integration.
connectionType	<code>string</code>	The integration connection type. The valid value is "VPC_LINK" for private integration or "INTERNET", otherwise.
credentials	<code>string</code>	For AWS IAM role-based credentials, specify the ARN of an appropriate IAM role. If unspecified, credentials will 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, please ensure that AWS STS regional endpoints are enabled for the region where this API is deployed for best performance.
contentHandling	<code>string</code>	Request payload encoding conversion types. Valid values are 1) <code>CONVERT_TO_TEXT</code> , 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) <code>CONVERT_TO_BINARY</code> , for converting a text payload into Base64-decoded blob or passing through a binary payload natively without modification.
httpMethod	<code>string</code>	The HTTP method used in the integration request. For Lambda

Property Name	Type	Description
		function invocations, the value must be <code>POST</code> .
<code>passthroughBehavior</code>	<code>string</code>	Specifies how a request payload of unmapped content type is passed through the integration request without modification. Supported values are <code>when_no_templates</code> , <code>when_no_match</code> , and <code>never</code> . For more information, see Integration.passthroughBehavior .
<code>requestParameters</code>	x-amazon-apigateway-integration.requestParameters Object (p. 500)	Specifies mappings from method request parameters to integration request parameters. Supported request parameters are <code>querystring</code> , <code>path</code> , <code>header</code> , and <code>body</code> .
<code>requestTemplates</code>	x-amazon-apigateway-integration.requestTemplates Object (p. 499)	Mapping templates for a request payload of specified MIME types.
<code>responses</code>	x-amazon-apigateway-integration.responses Object (p. 501)	Defines the method's responses and specifies desired parameter mappings or payload mappings from integration responses to method responses.
<code>timeoutInMillis</code>	<code>integer</code>	Integration timeouts between 50 ms and 29,000 ms.

Property Name	Type	Description
type	string	<p>The type of integration with the specified backend. The valid value is</p> <ul style="list-style-type: none"> • <code>http</code> or <code>http_proxy</code>: for integration with an HTTP backend;; • <code>aws_proxy</code>: for integration with AWS Lambda functions; • <code>aws</code>: for integration with AWS Lambda functions or other AWS services, such as Amazon DynamoDB, Amazon Simple Notification Service or Amazon Simple Queue Service; • <code>mock</code>: for integration with API Gateway without invoking any backend. <p>For more information about the integration types, see integration:type.</p>
uri	string	The endpoint URI of the backend. For integrations of the <code>aws</code> type, this is an ARN value. For the HTTP integration, this is the URL of the HTTP endpoint including the <code>https</code> or <code>http</code> scheme.

x-amazon-apigateway-integration Example

The following example integrates an API's `POST` method with a Lambda function in the backend. 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 ("") of the JSON string in the mapping templates must be string-escaped (\").

x-amazon-apigateway-integration.requestTemplates Object

Specifies mapping templates for a request payload of the specified MIME types.

Properties

Property Name	Type	Description
<i>MIME type</i>	string	An example of the MIME type is application/json. For information about creating a mapping template, see Mapping Templates (p. 168) .

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

Specifies mappings from named method request parameters to integration request parameters. The method request parameters must be defined before being referenced.

Properties

Property Name	Type	Description
integration.request.<param-type>.<param-name>	string	The value must be a predefined method request parameter of the method.request.<param-type>.<param-name> format, where <param-type> can be querystring, path, header, or body. For the body parameter, the <param-name> is a JSON path expression without the \$. prefix.

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.

```
"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
<i>Response status pattern</i>	x-amazon-apigateway-integration.response Object (p. 502)	<p>Selection regular expression used to match the integration response to the method response. For HTTP integrations, this regex applies to the integration response status code. For Lambda invocations, the regex applies to the <code>errorMessage</code> field of the error information object returned by AWS Lambda as a failure response body when the Lambda function execution throws an exception.</p> <p>Note The <i>Response status pattern</i> property name refers to a response status code or regular expression describing a group of response status codes. It does not correspond to any identifier of an IntegrationResponse resource in the API Gateway REST API.</p>

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 Swagger Operation responses field.
responseTemplates	x-amazon-apigateway-integration.responseTemplates Object (p. 503)	Specifies MIME type-specific mapping templates for the response's payload.
responseParameters	x-amazon-apigateway-integration.responseParameters Object (p. 504)	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 modification, and 2) CONVERT_TO_BINARY, for converting a text payload into 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
<i>MIME type</i>	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 Mapping Templates (p. 168) . An example of the <i>MIME type</i> 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. Only the header and body types of the integration response parameters can be mapped to the header type of the method response.

Properties

Property Name	Type	Description
method.response.header.< <i>string name</i> >		The named parameter value can be derived from the header and body types of the integration response parameters only.

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-request-validator Property

Specifies a request validator, by referencing a [request_validator_name](#) of the [x-amazon-apigateway-requestValidators Object \(p. 505\)](#) 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.

```
{
  "swagger": "2.0",
  "x-amazon-apigateway-requestValidators" : {
    "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-requestValidators 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 an API.

Properties

Property Name	Type	Description
<code>request_validator_name</code>	x-amazon-apigateway-requestValidators.requestValidator Object (p. 506)	<p>Specifies the validation rules consisting of the named validator. For example:</p> <div style="border: 1px solid black; padding: 10px;"> <pre>"basic" : { "validateRequestBody" : true, "validateRequestParameters" : true },</pre> </div> <p>To apply this validator to a specific method, reference the validator name (basic) as the value of the x-amazon-apigateway-request-validator Property (p. 504) property.</p>

x-amazon-apigateway-requestValidators 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.

```
{
  "swagger": "2.0",
  ...
  "x-amazon-apigateway-requestValidators" : {
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  ...
}
```

x-amazon-apigateway-requestValidators.requestValidator Object

Specifies the validation rules of a request validator as part of the [x-amazon-apigateway-requestValidators Object \(p. 505\)](#) map definition.

Properties

Property Name	Type	Description
validateRequestBody	Boolean	Specifies whether to validate the request body (<code>true</code>) or not (<code>false</code>).
validateRequestParameters	Boolean	Specifies whether to validate the required request parameters (<code>true</code>) or not (<code>false</code>).

x-amazon-apigateway-requestValidators.requestValidator Example

The following example shows a parameter-only request validator:

```
"params-only": {
  "validateRequestBody" : false,
  "validateRequestParameters" : true
}
```

Samples and Tutorials

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

Topics

- [Create an API Gateway API for AWS Lambda Functions \(p. 507\)](#)
- [Create an API as an Amazon S3 Proxy \(p. 526\)](#)
- [Create an API Gateway API as an Amazon Kinesis Proxy \(p. 551\)](#)

Create an API Gateway API for AWS Lambda Functions

Note

To integrate your API Gateway API with Lambda, you must choose a region where both the API Gateway and Lambda services are available. For region availability, see [Regions and Endpoints](#).

In the [Getting Started \(p. 18\)](#) section, you learned how to use the API Gateway console to build an API to expose a Lambda function. There, the console let you choose `Lambda Function` for **Integration type**, among other options of `HTTP`, `Mock` and `AWS Service`. The `Lambda Function` option is a special case of the `AWS Service` integration type and simplifies the integration set-up for you with default settings. For example, with the former, the console automatically adds the required resource-based permissions for invoking the Lambda function. With the latter, you have more control, but more responsibilities to set up the integration, including creating and specifying an IAM role containing appropriate permissions. For the both options, the underlying `integration.type` is `AWS` in the API Gateway REST API and its Swagger definition file.

In this section, we walk you through the steps to integrate an API with a Lambda function using the `AWS Service` and `Lambda Function` integration types. To support asynchronous invocation of the Lambda function, you must explicitly add the `X-Amz-Invocation-Type: Event` header to the integration request. For the synchronous invocation, you can add the `X-Amz-Invocation-Type: RequestResponse` header to the integration request or leave it unspecified. The following example shows the integration request of an asynchronous Lambda function invocation:

```
POST /2015-03-31/functions/FunctionArn/invocations?Qualifier=Qualifier HTTP/1.1
X-Amz-Invocation-Type: Event
...
Authorization: ...
Content-Type: application/json
Content-Length: PayloadSize

Payload
```

In this example, `FunctionArn` is the ARN of the Lambda function to be invoked. The `Authorization` header is required by secure invocation of Lambda functions over HTTPS. For more information, see the [Invoke](#) action in the [AWS Lambda Developer Guide](#).

To illustrate how to create and configure an API as an AWS service proxy for Lambda, we will create a Lambda function (`Calc`) that performs addition (+), subtraction (-), multiplication (*), and division (/).

When a client submits a method request to perform any of these operations, API Gateway will post the corresponding integration request to call the specified Lambda function, passing the required input (two operands and one operator) as a JSON payload. A synchronous call will return the result, if any, as the JSON payload. An asynchronous call will return no data.

You can expose a GET or POST method on the /calc resource to invoke the Lambda function. With the GET method, a client supplies the input to the backend Lambda function through three query string parameters (`operand1`, `operand2`, and `operator`). You will set up a mapping template to map these to the JSON payload of the integration request. With the POST method, a client provides the input to the Lambda function as a JSON payload of the method request. You can pass the method request payload through to the integration request, if the client input conforms to the input model. Alternatively, you can expose a GET method on the /calc/{`operand1`}/{`operand2`}/{`operator`} resource. With this method, the client specifies the Lambda function input as the values of the path parameters. You will need to provide a mapping template to translate the path parameters of the method request into an integration request payload as the Lambda function input and to translate the output from the integration responses to the method response.

In this tutorial, we will cover the following topics:

- Create the Calc Lambda function to implement the arithmetic operations, accepting and returning JSON-formatted input and output.
- Expose GET on the /calc resource to invoke the Lambda function, supplying the input as query strings. We will enable a request validator to ensure that the client submit all the required query string parameters before API Gateway calling the Lambda function.
- Expose POST on the /calc resource to invoke the Lambda function, supplying the input in the payload. We will enable a request validator to ensure that the client submitted the valid request payload before API Gateway call the Lambda function.
- Expose GET on the /calc/{`operand1`}/{`operand2`}/{`operator`} resource to invoke the Lambda function, specifying the input in the path parameters. We also explain how to define a Result schema to model the method response body so that any strongly typed SDK of the API can access the method response data through properties defined in the Result schema.

You can inspect the sample API in its [Swagger definition file \(p. 521\)](#). You can also [import](#) the API Swagger definitions to API Gateway, following the instructions given in [Import an API into API Gateway \(p. 238\)](#).

To use the API Gateway console to create the API, you must first sign up for an AWS account.

If you do not have an AWS account, use the following procedure to create one.

To sign up for AWS

1. Open <https://aws.amazon.com/> and choose **Create an AWS Account**.
2. Follow the online instructions.

To allow the API to invoke Lambda functions, you must have an IAM role that has appropriate IAM policies attached to it. The next section describes how to verify and to create, if necessary, the required IAM role and policies.

Topics

- [Set Up an IAM Role and Policy for an API to Invoke Lambda Functions \(p. 509\)](#)
- [Create a Lambda Function in the Backend \(p. 509\)](#)
- [Create API Resources for the Lambda Function \(p. 511\)](#)
- [Create a GET Method with Query Parameters to Call the Lambda Function \(p. 511\)](#)

- [Create a POST Method with a JSON Payload to Call the Lambda Function \(p. 515\)](#)
- [Create a GET Method with Path Parameters to Call the Lambda Function \(p. 517\)](#)
- [Swagger Definitions of Sample API Integrated with a Lambda Function \(p. 521\)](#)

Set Up an IAM Role and Policy for an API to Invoke Lambda Functions

For API Gateway to invoke a Lambda function, the API must have a permission to call the Lambda's [InvokeFunction](#) action. This means that, at minimum, you must attach the following IAM policy to an IAM role for API Gateway to assume the policy.

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

If you do not enact this policy, the API caller will receive a 500 Internal Server Error response. The response contains the "Invalid permissions on Lambda function" error message. For a complete list of error messages returned by Lambda, see the [Invoke](#) topic.

An API Gateway assumable role 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"  
        }  
    ]  
}
```

Create a Lambda Function in the Backend

The following procedure outlines the steps to create a Lambda function using the Lambda console.

1. Go to the Lambda console.
2. Choose **Create a Lambda function**.
3. Choose the **Blank Function** blueprint for the runtime of Node.js 4.3 or later.
4. Follow the on-screen instructions to the **Lambda function code** editor.
5. Copy the following Lambda function and paste it into the code editor in the Lambda console.

```

console.log('Loading the Calc function');

exports.handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    if (event.a === undefined || event.b === undefined || event.op === undefined) {
        callback("400 Invalid Input");
    }

    var res = {};
    res.a = Number(event.a);
    res.b = Number(event.b);
    res.op = event.op;

    if (isNaN(event.a) || isNaN(event.b)) {
        callback("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":
            res.c = res.b==0 ? NaN : Number(event.a) / Number(event.b);
            break;
        default:
            callback("400 Invalid Operator");
            break;
    }
    callback(null, res);
};

```

6. Choose an existing or create a new IAM role
7. Follow the on-screen instructions to finish creating the function.

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.

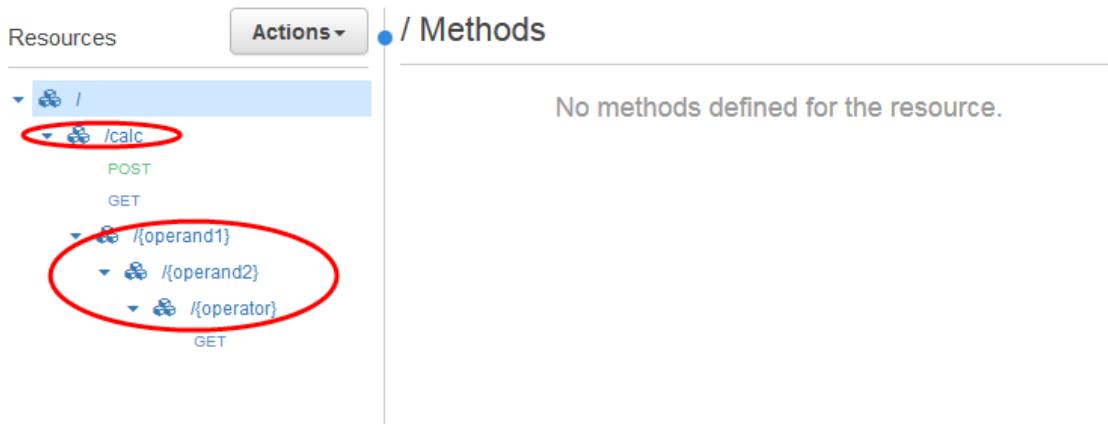
Create API Resources for the Lambda Function

The following procedure describes how to create API resources for the Lambda function. As an illustration, we use multiple API resources and methods to enable different API behaviors for calling the same function.

To create API resources for Lambda functions

1. In the API Gateway console, create an API named **LambdaGate**.
2. Create the **/calc** resource off the API's root. We will expose the GET and POST methods on this resource for the client to invoke the backend Lambda function. The caller must supply the required input as query string parameters (to be declared as ?operand1=...&operand2=...&operator=...) in the GET request and as a JSON payload in the POST request, respectively.

We will also create the **/calc/{operand1}/{operand2}/{operator}** resource subtree to expose the GET method to invoke the Lambda function. The caller must supply the required input by specifying the three path parameters (**operand1**, **operand2**, and **operator**).



Create a GET Method with Query Parameters to Call the Lambda Function

By creating a GET method with query parameters to call the Lambda function, we can let the API user to do the calculations via any browser. This can be useful especially if the API allows open access.

To set up the GET method with query strings to invoke the Lambda function

1. In the API Gateway console, choose the API's **/calc** resource under **Resources**.
2. Choose **Create Method**, from the **Actions** drop-down menu, to create the GET method.
3. In the ensuing **Set up** pane,

- a. Choose AWS Service for **Integration type**.
- b. Choose the region (e.g., us-west-2) where you created the Lambda function for **AWS Region**.
- c. Choose Lambda for **AWS Service**.
- d. Leave **AWS Subdomain** blank because our Lambda function is not hosted on any of AWS subdomain.
- e. Choose POST for **HTTP method**. Lambda requires that the POST request be used to invoke any Lambda function. This examples shows that the HTTP method in a frontend method request can be different from the integration request in the backend.
- f. Choose Use path override for **Action Type**. This option allows us to specify the ARN of the **Invoke** action to execute our Calc function.
- g. Type /2015-03-31/functions/arn:aws:lambda:**region:account-id:function:Calc/invocations** in **Path override**.
- h. Specify the ARN of an IAM role for **Execution role**. You can find the ARN of the role in the IAM console. The role must contain the necessary permissions for the caller to call the Calc function and for API Gateway to assume the role of the caller.
- i. Leave the Passthrough as the default of **Content Handling**, because we will not deal with any binary data.
- j. Choose the **Save** to finish setting up the GET /calc method.

After the setup succeeds, the configuration should look as follows:

[Method Execution](#) /calc - GET - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function 
 HTTP 
 Mock 
 AWS Service 

AWS Region us-west-2 

AWS Service Lambda 

AWS Subdomain 

HTTP method POST 

Path override /2015-03-31/functions/arn:aws:lambda:us-west-2:**7:function:Calc/invocations** 

Execution role arn:aws:iam::7  7:role/apigAwsProxyRole 

Credentials cache Do not add caller credentials to cache key 

Content Handling Passthrough  

You can also add, in **Integration Request**, the X-Amz-Invocation-Type: Event | RequestReponse | DryRun header to have the action invoked asynchronously, as request and response, or as a test run, respectively. If the header is not specified, the action will be invoked as request and response.

4. Go to **Method Request** to set up query parameters for the **GET** method on **/calc** to receive input to the backend Lambda function.

- a. Choose the pencil icon next to **Request Validator** to select Validate query string parameters and headers. This setting will cause an error message to return to state the required parameters are missing if the client does not specify them. You will not get charged for the call to the backend.
- b. Expand the **URL Query String Parameters** section. Choose **Add query string** to add the **operand1**, **operand2**, and **operator** query string parameters. Check the **Required** option for each parameter to ensure that they are validated.

The configuration now looks as follows:

Method Execution /calc - GET - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization: NONE

Request Validator: Validate query string parameters and headers

API Key Required: false

URL Query String Parameters

Name	Required	Caching
operand1	<input checked="" type="checkbox"/>	<input type="checkbox"/>
operand2	<input checked="" type="checkbox"/>	<input type="checkbox"/>
operator	<input checked="" type="checkbox"/>	<input type="checkbox"/>

+ Add query string

HTTP Request Headers

Request Body

SDK Settings

5. Go back to **Integration Request** to set up the mapping template to translate the client-supplied query strings to the integration request payload as required by the Calc function.
 - a. Expand the **Body Mapping Templates** section.
 - b. Choose When no template matches the request Content-Type header for **Request body passthrough**.
 - c. If application/json is not shown under **Content-Type**, choose **Add mapping template** to add it.
 - d. And then type and save the following mapping script in the mapping template editor:

```
{
    "a": "$input.params('operand1')",
}
```

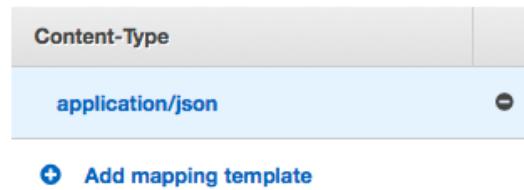
```
"b": "$input.params('operand2')",
"op": "$input.params('operator')"
}
```

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.

The final settings of this step is shown as follows:

▼ Body Mapping Templates

- Request body passthrough When no template matches the request Content-Type header  
 When there are no templates defined (recommended) 
 Never 



application/json

Generate template:

```
1 {
2   "a": "$input.params('operand1')",
3   "b": "$input.params('operand2')",
4   "op": "$input.params('operator')"
5 }
```

6. You can now choose **Test** to verify that the GET method on the `/calc` resource has been properly set up to invoke the Lambda function.

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, we expect the client to submit the necessary input to the backend function in the request body. To ensure that the client uploads the correct input data, we will enable request validation on the payload.

To set up the POST method with a JSON payload to invoke a Lambda function

1. Go to the API Gateway console and choose the API created previously.
2. Highlight the **/calc** resource from **Resources** pane.
3. Choose **Create Method** from the **Actions** menu to create the **POST /calc** method.
4. In the method's **Set Up** panel, configure this POST method with the following integration settings. For more information, follow the discussions in [Create a GET Method with Query Parameters to Call the Lambda Function \(p. 511\)](#).

[Method Execution](#) /calc - POST - Integration Request 

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function 
 HTTP 
 Mock 
 AWS Service 

AWS Region us-west-2 

AWS Service Lambda 

AWS Subdomain 

HTTP method POST 

Path override /2015-03-31/functions/arn:aws:lambda:us-west-2:7:7:function:Calc/invocations 

Execution role arn:aws:iam::7:role/apigAwsProxyRole 

Credentials cache Do not add caller credentials to cache key 

Content Handling Passthrough 

5. Choose **Models** under the API from the API Gateway console's primary navigation pane to create data models for the method's input and output:

- a. Choose **Create** in the **Models** pane. Type **Input** in **Model name**, type **application/json** in **Content type**, and type the following schema definition in **Model schema**:

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

```
}
```

This model describes the input data structure and will be used to validate the incoming request body.

- b. Choose **Create** in the **Models** pane. Type **Output** in **Model name**, type **application/json** in **Content type**, and type the following schema definition in **Model schema**:

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

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.

- c. Choose **Create** in the **Models** pane. Type **Result** in **Model name**, type **application/json** in **Content type**, and type the following schema definition in **Model schema**:

```
{
    "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": "Output"
}
```

This model describes the data structure of the returned response data. It references both the **Input** and **Output** schemas defined in the specified API (*restapi-id*). Again, this model is not used in this tutorial because it leverages the passthrough behavior.

6. In the **Method Request** configuration settings, do the following to enable request validation on the incoming request body:
 - a. Choose the pencil icon next to **Request Validator** to choose **Validate body**.
 - b. Expand the **Request Body** section, choose **Add model**
 - c. Type **application/json** in the **Content-Type** input field and choose **Input** from the drop-down list in the **Model name** column.
7. You can now choose **Test** to verify the POST method works as expected. The following input:

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

should produce the following output:

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

If you would like to implement this method as an asynchronous call, you can add an `InvocationType` header in the method request and map it to the `X-Amz-Invocation-Type` header in the integration request with either a static value of '`Event`' or the header mapping expression `method.request.header.InvocationType`. For the latter, the client must include the `InvocationType:Event` header in the method request. The asynchronous call will return an empty response, instead.

Create a GET Method with Path Parameters to Call the Lambda Function

In this section, we 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. We will define a mapping template to map the incoming path parameter values to the required integration request payload.

In addition, we will use the simple Lambda integration feature provided by the API Gateway console to set up the method. As you can see, this console-provided feature provides much more streamlined user experiences.

To set up the GET method with URL path parameters to call the Lambda function

1. Go to the API Gateway console.
2. Highlight the `/calc/{operand1}/{operand2}/{operator}` resource on the **Resources** tree of the previously created API.
3. Choose **Create Method** from the **Actions** drop-down menu, choose **GET**.
4. In the **Setup** pane, choose **Lambda Function** for **Integration type**, to use the streamlined setup process enabled by the console.
5. Choose a region (e.g., `us-west-2`) for **Lambda Region**. This is the region where the Lambda function is hosted.
6. Choose an existing Lambda function (e.g., `Calc`) for **Lambda Function**.
7. Choose **Save** and then choose **OK** to consent to **Add Permissions to Lambda Function**.
8. Choose **Integration Request** to set up body mapping template.
 - a. Expand the **Body Mapping Templates** section.
 - b. Choose **Add mapping template**.
 - c. Type `application/json` for **Content-Type** and then choose the check mark icon to open the template editor.
 - d. Choose **Yes, secure this integration** to proceed.
 - e. Type the following mapping script to the template editor:

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

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

This template maps the three URL path parameters, declared when the **/calc/{operand1}/{operand2}/{operator}** resource was created, into designated property values of 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.

- f. Save the mapping template.

When the method is set up correctly, the settings should look similar to the following:

[Method Execution](#) /calc/{operand1}/{operand2}/{operator} - GET - Integration...

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function [i](#)
 HTTP [i](#)
 Mock [i](#)
 AWS Service [i](#)

Use Lambda Proxy integration [i](#)

Lambda Region us-west-2 [e](#)

Lambda Function Calc [e](#)

Invoke with caller credentials [i](#)

Credentials cache Do not add caller credentials to cache key [e](#)

▼ Body Mapping Templates

Request body passthrough When no template matches the request Content-Type header [i](#)
 When there are no templates defined (recommended) [i](#)
 Never [i](#)

Content-Type
application/json

[+ Add mapping template](#)

application/json

Generate template: [e](#)

```
1 <!
2   "a": "$input.params('operand1')",
3   "b": "$input.params('operand2')",
4   "op": "#if($input.params('operator')=='%2F')/#{else}{$input.params('operator')}#end"
5
6 >
```

9. Now, let us test the API using the TestInvoke feature of the console.

- a. Choose **Test** from **Method Execution**.
- b. Type 1, 2 and + in **{operand1}**, **{operand2}** and **{operator}** fields, respectively.
- c. Choose **Test**.
- d. The result will be shown similar to the following:

Request: /calc/1/1/+

Status: 200

Latency: 816 ms

Response Body

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

Response Headers

```
{"X-Amzn-Trace-Id":"sampled=0;root=1-58f7f1f6-57c26581ed7073a711c13216","Content-Type":"application/json"}
```

Logs

```
Execution log for request test-request  
Wed Apr 19 23:25:42 UTC 2017 : Starting execution for request: test-invoke-request  
Wed Apr 19 23:25:42 UTC 2017 : HTTP Method: GET, Resource Path: /calc/1/1/  
Wed Apr 19 23:25:42 UTC 2017 : Method request path: {operand1=1, operand2=1, operator =+}  
Wed Apr 19 23:25:42 UTC 2017 : Method request query string: {}  
Wed Apr 19 23:25:42 UTC 2017 : Method request headers: {}  
Wed Apr 19 23:25:42 UTC 2017 : Method request body before transformations:  
Wed Apr 19 23:25:42 UTC 2017 : Endpoint request URI: https://lambda.us-west-2.amazonaws.com/2015-03-31/functions/arn:aws:lambda:us-west-2:738575810317:function:Calc/invocations  
Wed Apr 19 23:25:42 UTC 2017 : Endpoint request headers: {x-amzn-lambda-integration-tag=test-request, Authorization=*****}
```

This test result shows the original output from the backend Lambda function, as passed through the integration response without mapping, because we have not configured any mapping template. Next, we model the data structure of the method response payload after the Result schema.

10. By default, the method response body is assigned an Empty model. This will cause the integration response body 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 both the REST client and SDK clients receive the desired result, you must model the response data using a predefined schema. Here, we demonstrate how to define a model for the method response body and to construct a mapping template to transform the integration response body to the method response body.
 - a. In **/calc/{operand1}/{operand2}/{operator} - GET - Method Execution**, choose **Method Response**.
 - b. Expand the **200** response,
 - c. Under **Response Body for 200** section. If no model has been assigned for the method response,
 - d. Choose the pencil icon next to the model for the **application/json** content type.
 - e. Choose a predefined model from the **Models** drop-down list. For this tutorial, this is **Result**.

- f. Save the model choice.

Note

If no model is defined for the content type of `application/json`, choose **Add Response Model** and follow the on-screen instructions to add the model.

Setting the model for the method response body ensure that the response data will be cast into the `Result` object of a given SDK. For this, we also need to make sure that the integration response data is mapped accordingly, which we discuss next.

11. To return the backend result according to the specified schema,
 - a. Choose **Integration Response** and expand the 200 method response entry.
 - b. Expand the **Body Mapping Templates** section.
 - c. Choose or add `application/json` to the **Content-Type** list.
 - d. Choose `Result` from the **Generate template** drop-down list to bring up the `Result` template blueprint.
 - e. Change the template blueprint as follows:

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

- f. Choose **Save**.
- g. To test the mapping template, choose **Test in Method Execution** and type `1 2` and `+` in the **operand1**, **operand2** and **operator** input fields, respectively. The integration response from the Lambda function is now mapped to a `Result` object:

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

12. To make the API accessible beyond Test Invoke in the API Gateway console, choose **Deploy API** from the **Actions** drop-down menu. Make sure to repeat deploying the API whenever you finish adding, modifying or deleting a resource or method, updating any data mapping, updating the stage settings. Otherwise, new features or updates will not be available.

Swagger Definitions of Sample API Integrated with a Lambda Function

```
{  
    "swagger": "2.0",  
    "info": {  
        "version": "2017-04-20T04:08:08Z",  
        "title": "LambdaGate"  
    },  
    "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",  
                                "headers": {  
                                    "Content-Type": "application/json"  
                                },  
                                "body": "The result of the calculation is {{ operand1 }} {{ operator }} {{ operand2 }}."  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    "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": [
        {
            "name": "InvocationType",
            "in": "header",
            "required": false,
            "type": "string"
        },
        {
            "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": {
                    "$ref": "#/definitions/Result"
                }
            }
        },
        "x-amazon-apigateway-integration": {
            "responses": {
                "default": {
                    "statusCode": "200",
                    "responseTemplates": {
                        "application/json": "#set($inputRoot = $input.path('$'))\n{\n    \"input\" : ${\n        \"a\" : $inputRoot.a,\n        \"b\" : $inputRoot.b,\n        \"op\" : \"$inputRoot.op\"\n    },\n    \"output\" : {\n        \"c\" : $inputRoot.c\n    }\n}\n"
                    }
                }
            },
            "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
            "passthroughBehavior": "when_no_templates",
            "httpMethod": "POST",
            "requestTemplates": {
                "application/json": "{\n        \"a\" : \"$input.params('operand1')\", \n        \"b\" :\n        \"$input.params('operand2')\", \n        \"op\" : #if($input.params('operator')=='%2F')/\n        '#{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-requestValidators": {
    "Validate body": {
        "validateRequestParameters": false,
        "validateRequestBody": true
    },
    "Validate query string parameters and headers": {
        "validateRequestParameters": true,
        "validateRequestBody": false
    }
}
}
```

Create an API as an Amazon S3 Proxy

As an example to showcase using an API in API Gateway to proxy Amazon S3, this section describes how to create and configure an 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 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](#).

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 [Regions and Endpoints](#).

You may want to import the sample API as an Amazon S3 proxy, as shown in [Swagger Definitions of the Sample API as an Amazon S3 Proxy \(p. 542\)](#). For instructions on how to import an API using the Swagger definition, see [Import an API into API Gateway \(p. 238\)](#).

To use the API Gateway console to create the API, you must first sign up for an AWS account.

If you do not have an AWS account, use the following procedure to create one.

To sign up for AWS

1. Open <https://aws.amazon.com/> and choose **Create an AWS Account**.
2. Follow the online instructions.

Topics

- [Set Up IAM Permissions for the API to Invoke Amazon S3 Actions \(p. 526\)](#)
- [Create API Resources to Represent Amazon S3 Resources \(p. 528\)](#)
- [Expose an API Method to List the Caller's Amazon S3 Buckets \(p. 529\)](#)
- [Expose API Methods to Access an Amazon S3 Bucket \(p. 534\)](#)
- [Expose API Methods to Access an Amazon S3 Object in a Bucket \(p. 537\)](#)
- [Call the API Using a REST API Client \(p. 540\)](#)
- [Swagger Definitions of the Sample API as an Amazon S3 Proxy \(p. 542\)](#)

Set Up IAM Permissions for the API to Invoke Amazon S3 Actions

To allow the API to invoke required Amazon S3 actions, you must have appropriate IAM policies attached to an IAM role. The next section describes how to verify and to create, if necessary, the required IAM role and policies.

For your API to view or list Amazon S3 buckets and objects, you can use the IAM-provided AmazonS3ReadOnlyAccess policy in the IAM role. The ARN of this policy is `arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess`, which is as shown as follows:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "s3:Get*",  
                "s3>List*"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

This policy document states that any of the Amazon S3 `Get*` and `List*` actions can be invoked on any of the Amazon S3 resources.

For your API to update Amazon S3 buckets and objects, you can use a custom policy for any of the Amazon S3 `Put*` actions as shown as follows:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "s3:Put*",  
            "Resource": "*"  
        }  
    ]  
}
```

For your API to work with Amazon S3 `Get*`, `List*` and `Put*` actions, you can add the above read-only and put-only policies to the IAM role.

For your API to invoke the Amazon S3 `Post*` actions, you must use an Allow policy for the `s3:Post*` actions in the IAM role. For a complete list of Amazon S3 actions, see [Specifying Amazon S3 Permissions in a Policy](#).

For your API to create, view, update, and delete buckets and objects in Amazon S3, you can use the IAM -provided AmazonS3FullAccess policy in the IAM role. The ARN is `arn:aws:iam::aws:policy/AmazonS3FullAccess`.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "s3:*",  
            "Resource": "*"  
        }  
    ]  
}
```

Having chosen the desired IAM policies to use, create an IAM role and attach to it the policies. The resulting IAM role must contain the following trust policy for API Gateway to assume this role at runtime.

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

When using the IAM console to create the role, choose the **Amazon API Gateway** role type to ensure that this trust policy is automatically included.

Create API Resources to Represent Amazon S3 Resources

We will use the API's root (/) resource as the container of an authenticated caller's Amazon S3 buckets. We will 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.

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

1. In the API Gateway console, create an API named **MyS3**. This API's root resource (/) represents the Amazon S3 service.
2. Under the API's root resource, create a child resource named **Folder** and set the required **Resource Path** as **/{folder}**.
3. For the API's **Folder** resource, create an **Item** child resource. Set the required **Resource Path** as **/ {item}**.

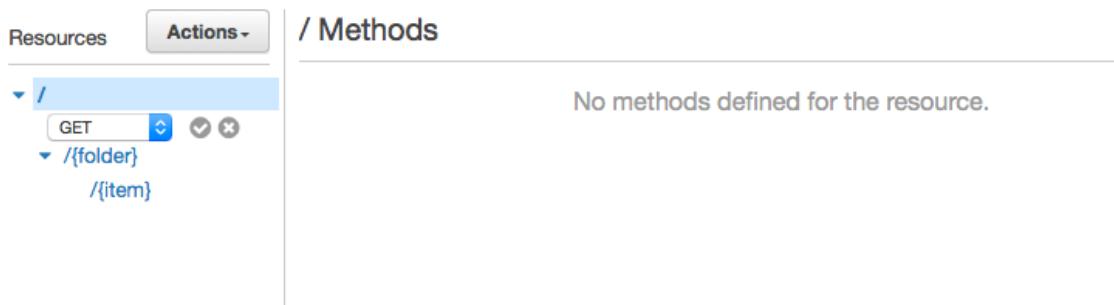
The screenshot shows the 'New Child Resource' dialog in the AWS API Gateway console. On the left, a sidebar shows a tree structure with a node labeled '/{folder}' highlighted with a red oval. The main panel has fields for 'Resource Name*' (set to 'Item') and 'Resource Path*' (set to '/{folder}/ {item}'). A note below explains path parameters. At the bottom, there are 'Enable API Gateway CORS' and 'Create Resource' buttons.

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. Choose **Create method** on the root node (/) from the **Actions** drop-down menu at the top-right corner of the **Resources** panel.
2. Choose the **GET** from the drop-down list of HTTP verbs, and choose the check-mark icon to start creating the method.



3. In the **/ - GET - Setup** pane, choose **AWS Service** for **Integration type**.
4. From the list, choose a region (e.g., us-west-2) for **AWS Region**.
5. From **AWS Service**, choose **S3**.
6. For **AWS Subdomain**, leave it blank.
7. From **HTTP method**, choose **GET**.
8. For **Action Type**, choose **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. (Optional) In **Path override** type **/**.
10. Copy the previously created IAM role's ARN (from the IAM console) and paste it into **Execution role**.
11. Leave any other settings as default.
12. Choose **Save** to finish setting up this method.

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

Note

After the initial setup, you can modify these settings in the **Integration Request** page of the method.

To control who can call this method of our API, we turn on the method authorization flag and set it to **AWS_IAM**.

To enable IAM to control access to the GET / method

1. From the **Method Execution**, choose **Method Request**.

2. Choose the pencil icon next to **Authorization**
3. Choose **AWS_IAM** from the drop-down list.
4. Choose the check-mark icon to save the setting.

[Method Execution](#) / - GET - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Authorization Settings



- ▶ URL Query String Parameters
- ▶ HTTP Request Headers
- ▶ Request Models [Create a Model](#)

For our API to return successful responses and exceptions properly to the caller, let us declare the 200, 400 and 500 responses in **Method Response**. We 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. From the **Method Execution** pane, choose the **Method Response** box. The API Gateway declares the 200 response by default.
2. Choose **Add response**, enter **400** in the input text box, and choose the check-mark to finish the declaration.
3. Repeat the above step to declare the 500 response type. The final setting is shown as follows:

[Method Execution](#) / - GET - Method Response

Provide information about this method's response types, their headers and content types.

	HTTP Status	
▶	200	X
▶	400	X
▶	500	X
Add 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, we 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, we 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. In the API Gateway console, choose **Method Response**. Add the **Content-Type** header for the 200 response type.

[← Method Execution](#) / - GET - Method Response

Provide information about this method's response types, their headers and content types.

HTTP Status
200

Response Headers for 200

Name
Timestamp
Content-Length
Content-Type

+ Add Header

Response Models for 200

Content type	Models
application/json	Empty

Create a model

+ Add Response Model

▶ 400

▶ 500

+ Add Response

2. In **Integration Response**, for **Content-Type**, type `integration.response.header.Content-Type` for the method response.

[Method Execution](#) / - GET - Integration Response

First, declare response types using [Method Response](#). Then, map the possible responses from the backend to this method's response types.

	HTTP status regex	Method response status	Output model	Default mapping	
▼	-	200		Yes	✖

Map the output from your HTTP endpoint to the headers and output model of the 200 method response.

HTTP status regex ⓘ

Method response status

[Cancel](#) [Save](#)

▼ Header Mappings

Response header	Mapping value ⓘ	
Timestamp	integration.response.header.Date	✎
Content-Length	integration.response.header.Content-Length	✎
Content-Type	integration.response.header.Content-Type	✎

▶ Body Mapping Templates

With the above header mappings, API Gateway will translate the Date header from the backend to the Timestamp header for the client.

3. Still in **Integration Response**, choose **Add integration response**, type an appropriate regular expression in the **HTTP status regex** text box for a remaining method response status. Repeat until all the method response status are covered.

[Method Execution](#) / - GET - Integration Response

First, declare response types using [Method Response](#). Then, map the possible responses from the backend to this method's response types.

	HTTP status regex	Method response status	Output model	Default mapping	
▶	-	200		Yes	✖
▶	4\d{2}	400		No	✖
▼	5\d{2}	500		No	✖

Map the output from your HTTP endpoint to the headers and output model of the 500 method response.

HTTP status regex 5\d{2} i

Method response status 500

Cancel **Save**

▼ Header Mappings

Response header	Mapping value i
No method response headers.	

▶ Body Mapping Templates

Add integration response +

As a good practice, let us test our API we have configured so far.

Test the GET method on the API root resource

1. Go back to **Method Execution**, choose **Test** from the **Client** box.
2. Choose **Test** in the **GET / - Method Test** pane. An example result is shown as follows.

[Method Execution](#) / - GET - Method Test

Make a test call to your method with the provided input

Path

No path parameters exist for this resource. You can define path parameters by using the syntax `{myPathParam}` in a resource path.

Query Strings

No query string parameters exist for this method. You can add them via Method Request.

Headers

No header parameters exist for this method. You can add them via Method Request.

Stage Variables

No stage variables exist for this method.

Client Certificate

None

Request Body

Request Body is not supported for GET methods.

 Test

Request: /

Status: 200

Latency: 746 ms

Response Body

```
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <Owner><ID>06e...</ID><DisplayName>aws...</DisplayName><Name>a...</Name><CreationDate>2016-02-12T22:05:55.000Z</CreationDate><Bucket><Name>a...</Name><CreationDate>2016-02-15T23:38:35.000Z</CreationDate><Bucket><Name>aws-devdoc-test-kd...</Name><CreationDate>2015-10-27T22:59:29.000Z</CreationDate><Bucket><Name>aws-lambda...</Name><CreationDate>2015-10-21T22:39:57.000Z</CreationDate><Bucket><Name>aws-lambda...</Name><CreationDate>2015-02-11T23:14:00.000Z</CreationDate><Bucket><Name>aws-lambda-triggers-k...</Name><CreationDate>2015-10-22T23:48:01.000Z</CreationDate><Bucket><Name>f-templates-1...</Name><CreationDate>2015-10-21T20:59:45.000Z</CreationDate><Bucket><Name>elasticbeanstalk-us-east-1...</Name><CreationDate>2015-10-16T23:13:15.000Z</CreationDate><Bucket><Name>jaws.dev....</Name><CreationDate>2015-12-01T19:34:56.000Z</CreationDate><Bucket><Name>ja...</Name><CreationDate>2015-12-01T20:22:47.000Z</CreationDate><Bucket><Name>m-pictures-02-16-2016/<Name><CreationDate>2016-02-18T08:14:40.000Z</CreationDate><Bucket><Bucket><Name>myrv-contentdelivery-mobilehub-1161765204/<Name><CreationDate>2015-10-26T05:07:51.000Z</CreationDate><Bucket><Bucket><Name>myrv-userfiles-mobilehub-1161765204/<Name><CreationDate>2015-10-26T05:09:30.000Z</CreationDate><Bucket><Name>node-sdk-sample-33358871-858c-40fd-8f90-9b0eeb096eee/<Name><CreationDate>2015-10-21T23:49:15.000Z</CreationDate></Bucket></Buckets></ListAllMyBucketsResult>
```

Response Headers

```
{"Timestamp": "Fri, 19 Feb 2016 03:09:52 GMT", "Content-Type": "application/xml"}
```

Logs

```
Execution log for request test-request
Fri Feb 19 03:09:50 UTC 2016 : Starting execution for request: test-invok...
```

Note

To use the API Gateway console to test the API as an Amazon S3 proxy, make sure that the targeted S3 bucket is from a different region from the API's region. Otherwise, you may get a 500 Internal Server Error response. This limitation does not apply to any deployed API.

Expose API Methods to Access an Amazon S3 Bucket

To work with an Amazon S3 bucket, we expose the GET, PUT, and DELETE methods on the `/{folder}` resource to list objects in a bucket, create a new bucket, and delete an existing bucket. The instructions are similar to those prescribed in [Expose an API Method to List the Caller's Amazon S3 Buckets \(p. 529\)](#). In the following discussions, we outline the general tasks and highlight relevant differences.

To expose GET, PUT and DELETE methods on a folder resource

1. On the `/{folder}` node from the **Resources** tree, create the DELETE, GET and PUT methods, one at a time.
2. Set up the initial integration of each created method with its corresponding Amazon S3 endpoints. The following screen shot illustrates this setting for the `PUT /{folder}` method. For the

`DELETE /{folder}` and `GET /{folder}` method, replace the `PUT` value of **HTTP method** by `DELETE` and `GET`, respectively.

The screenshot shows the 'PUT - Setup' configuration for a method named '{folder}'. The left sidebar lists other methods: GET, DELETE, and /{item}. The PUT method is currently selected. The main configuration area includes:

- Integration type:** AWS Service (selected)
- AWS Region:** <region>
- AWS Service:** S3 (highlighted with a red oval)
- AWS Subdomain:** (empty)
- HTTP method:** PUT (highlighted with a red oval)
- Action Type:** Use path override (selected)
- Path override (optional):** {bucket} (highlighted with a red oval)
- Execution role:** arn:aws:iam::...:role/apigAwsProxyRole
- Save** button

Notice that we used the `{bucket}` path parameter in the Amazon S3 endpoint URLs to specify the bucket. We will need to map the `{folder}` path parameter of the method requests to the `{bucket}` path parameter of the integration requests.

3. To map `{folder}` to `{bucket}`:
 - a. Choose **Method Execution** and then **Integration Request**.
 - b. Expand **URL Path Parameters** and choose **Add path**
 - c. Type `bucket` in the **Name** column and `method.request.path.folder` in the **Mapped from** column. Choose the check-mark icon to save the mapping.

▼ URL Path Parameters

Name	Mapped from	Caching
bucket	method.request.path.folder	<input checked="" type="checkbox"/>

4. In **Method Request**, add the **Content-Type** to the **HTTP Request Headers** section.

▼ HTTP Request Headers

Name	Caching	
Content-Type	<input type="checkbox"/>	 
 Add header		

This is mostly needed for testing, when using the API Gateway console, when you must specify **application/xml** for an XML payload.

5. In **Integration Request**, set up the following header mappings, following the instructions described in [Expose an API Method to List the Caller's Amazon S3 Buckets \(p. 529\)](#).

▼ HTTP Headers

Name	Mapped from 	Caching	
x-amz-acl	'authenticated-read'	<input type="checkbox"/>	 
Content-Type	method.request.header.Content-Type	<input type="checkbox"/>	 
 Add header			

The **x-amz-acl** header is for specifying access control on the folder (or the corresponding Amazon S3 bucket). For more information, see [Amazon S3 PUT Bucket Request](#).

6. To test the **PUT** method, choose **Test** in the **Client** box from **Method Execution**, and enter the following as input to the testing:
 - In **folder**, type a bucket name,
 - For the **Content-Type** header, type **application/xml**.
 - In **Request Body**, provide the bucket region as the location constraint, declared in an XML fragment as the request payload. For example,

```
<CreateBucketConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <LocationConstraint>{region}</LocationConstraint>
</CreateBucketConfiguration>
```

The screenshot shows the 'Method Execution' interface for the '/{folder}' - PUT - Method Test. The 'Path' field contains 'my-pictures-02-16-2016'. The 'Content-Type' header is set to 'application/xml'. The 'Request Body' contains XML for creating a bucket:

```

<?xml version="1.0" encoding="UTF-8"?>
<CreateBucketConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01">
  <LocationConstraint>us-west-2</LocationConstraint>
</CreateBucketConfiguration>

```

The response shows a status of 200, latency of 984 ms, and no data in the body. The logs show the execution log for the request.

7. Repeat the preceding steps to create and configure the GET and DELETE method on the API's / {folder} resource.

The above examples illustrate how to create a new bucket in the specified region, to view the list of objects in the bucket, and to delete the bucket. Other Amazon S3 bucket operations allow you work with the metadata or properties of the bucket. For example, you can set up your API to call the Amazon S3's [PUT /notification](#) action to set up notifications on the bucket, to call [PUT /acl](#) to set an access control list on the bucket, etc. The API set up is similar, except for that you must append appropriate query parameters to the Amazon S3 endpoint URLs. At run time, you must provide the appropriate XML payload to the method request. The same can be said about supporting the other GET and DELETE operations on a Amazon S3 bucket. For more information on possible &S3; actions on a bucket, see [Amazon S3 Operations on Buckets](#).

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. For the complete list of supported actions, see [Amazon S3 Operations on Objects](#).

In this tutorial, we expose the [PUT Object](#) operation, the [GET Object](#) operation, [HEAD Object](#) operation, and the [DELETE Object](#) operation through the API methods of `PUT /{folder}/{item}`, `GET /{folder}/{item}`, `HEAD /{folder}/{item}` and `DELETE /{folder}/{item}`, respectively.

The API setups for the PUT, GET and DELETE methods on `/ {folder} / {item}` are the similar to those on `/ {folder}`, as prescribed in [Expose API Methods to Access an Amazon S3 Bucket \(p. 534\)](#). One major difference is that the object-related request path has an additional path parameter of `{item}` and this path parameter must be mapped to the integration request path parameter of `{object}`.

[◀ Method Execution](#) /{folder}/{item} - PUT - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function [i](#)

HTTP [i](#)

Mock [i](#)

AWS Service [i](#)

AWS Region [u](#) 2 [e](#)

AWS Service S3 [e](#)

AWS Subdomain [e](#)

HTTP method PUT [e](#)

Path override `{bucket}/{object}` [e](#)

Execution role `arn:aws:iam::123456789012:role/apigAwsProxyRole` [e](#)

Credentials cache Do not add caller credentials to cache key [e](#)

▼ URL Path Parameters

Name	Mapped from i	Caching	e
object	<code>method.request.path.item</code>	<input type="checkbox"/>	e
bucket	<code>method.request.path.folder</code>	<input type="checkbox"/>	e

[+ Add path](#)

The same is true for the GET and DELETE methods.

As an illustration, the following screen shot shows the output when testing the GET method on a `{folder}/{item}` resource using the API Gateway console. The request correctly returns the plain text of ("Welcome to README.txt") as the content of the specified file (README.txt) in the given Amazon S3 bucket (apig-demo).

◀ Method Execution /{folder}/{item} - GET - Method Test

Make a test call to your method with the provided input

Path

folder

apig-demo

item

README.txt

Query Strings

No query string parameters exist for this method.
You can add them via Method Request.

Headers

No header parameters exist for this method. You
can add them via Method Request.

Stage Variables

No stage variables exist for this method.

Client Certificate

None

Request Body

Request Body is not supported for GET
methods.

Test

Request: /apig-demo/README.txt

Status: 200

Latency: 486 ms

Response Body

Welcome to README.txt

Response Headers

{"Content-Type": "text/plain"}

Logs

```
Execution log for request test-request
Fri Feb 19 03:35:20 UTC 2016 : Starting execution for request: test-invoke-request
Fri Feb 19 03:35:20 UTC 2016 : API Key: test-invoke-api-key
Fri Feb 19 03:35:20 UTC 2016 : Method request path: {item=README.txt, folder=apig-demo}
Fri Feb 19 03:35:20 UTC 2016 : Method request query string: {}
Fri Feb 19 03:35:20 UTC 2016 : Method request headers: {}
Fri Feb 19 03:35:20 UTC 2016 : Method request body before transformations: null
Fri Feb 19 03:35:20 UTC 2016 : Endpoint request URI: http://s3-us-west-2.amazonaws.com/apig-demo/README.txt
Fri Feb 19 03:35:20 UTC 2016 : Endpoint request headers: {Authorization=*****}
*****
*****
*****
*****
*****a99006, X-Amz
```

To download or upload binary files, which in API Gateway is considered anything 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 **Binary Support** for the API (from the API Gateway primary navigation panel),
 - b. Choose **Edit**.
 - c. Type the required media type (e.g., `image/png` for **Binary media types**).
 - d. Choose **Add binary media type** 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 the integration response (for download). Make sure that no mapping template is defined for the affected content type. For more information, see [Integration Passthrough Behaviors \(p. 190\)](#) and [Select VTL Mapping Templates \(p. 189\)](#).

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 \(p. 200\)](#).

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 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=ccadb877bdb0d395ca38c47e18a0d76bb5eaf17007d11e40bf6fb63d28c705b
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: d60fcfb9-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...603add12ee</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`.

Swagger Definitions of the Sample API as an Amazon S3 Proxy

The following Swagger definitions describe the sample API , referenced in this tutorial, as an Amazon S3 proxy.

```
{
  "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::<replaceable>123456789012</replaceable>:role/apigAwsProxyRole",
          "responses": {
            "4\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type": "'application/json'"
              }
            }
          }
        }
      }
    }
  }
}
```

```
        "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::<replaceable>123456789012</replaceable>:role/  
apigAwsProxyRole",  
            "responses": {  
                "4\\d{2}": {  
                    "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"  
                }  
            }  
        }  
    }  
}
```

```
        "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::<replaceable>123456789012</replaceable>: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.header.x-amz-acl": "'authenticated-read'",
    "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"
    }
  }
}

```

```
        "description": "500 response"
    }
},
"security": [
{
    "sigv4": []
}
],
"x-amazon-apigateway-integration": {
    "credentials": "arn:aws:iam::<replaceable>123456789012</replaceable>: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::<replaceable>123456789012</replaceable>: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::<replaceable>123456789012</replaceable>: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::<replaceable>123456789012</replaceable>: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.header.x-amz-acl": "'authenticated-read'",
          "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::<replaceable>123456789012</replaceable>: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"
    }
}
}

```

Create an API Gateway API as an Amazon Kinesis Proxy

This section describes how to create and configure an API Gateway 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 [Regions and Endpoints](#).

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 either the API Gateway [Import API](#) or the [API Gateway Swagger Importer](#). For information on how to use the Import API, see [Import an API into API Gateway \(p. 238\)](#). For information on how to use the API Gateway Swagger Importer, see [Getting Started with the API Gateway Swagger Importer](#).

If you do not have an AWS account, use the following procedure to create one.

To sign up for AWS

1. Open <https://aws.amazon.com/> and choose **Create an AWS Account**.
2. Follow the online instructions.

Create an IAM Role and Policy for the API to Access Kinesis

To allow the API to invoke Kinesis actions, you must have appropriate IAM policies attached to an IAM role. This section explains how to verify and to create, if necessary, the required IAM role and policies.

To enable read-only access to Kinesis, you can use the `AmazonKinesisReadOnlyAccess` policy that allows the `Get*`, `List*`, and `Describe*` actions in Kinesis to be invoked.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "kinesis:Get*",  
                "kinesis>List*",  
                "kinesis:Describe*"
```

```
        ],
        "Resource": "*"
    }
}
```

This policy is available as an IAM-provisioned managed policy and its ARN is [arn:aws:iam::aws:policy/AmazonKinesisReadOnlyAccess](#).

To enable read-write actions in Kinesis, you can use the [AmazonKinesisFullAccess](#) policy.

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

This policy is also available as an IAM-provisioned managed policy. Its ARN is [arn:aws:iam::aws:policy/AmazonKinesisFullAccess](#).

After you decide which IAM policy to use, attach it to a new or existing IAM role. Make sure that the API Gateway control service ([apigateway.amazonaws.com](#)) is a trusted entity of the role and is allowed to assume the execution role (`sts:AssumeRole`).

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

If you create the execution role in the IAM console and choose the **Amazon API Gateway** role type, this trust policy is automatically attached.

Note the ARN of the execution role. You will need it when creating an API method and setting up its integration request.

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. In the API Gateway console, choose **Create API**.

2. Choose **New API**.
3. In **API name**, type **KinesisProxy**. Leave the default values in the other fields.
4. Type a description in **Description**, if you like.
5. 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 API root resource. In **Actions**, choose **Create Resource**.

In **Resource Name**, type `Streams`, leave **Resource Path** and other fields as the default, and choose **Create Resource**.

2. Choose the `/Streams` resource. From **Actions**, choose **Create Method**, choose `GET` from the list, and then choose the check mark icon to finish creating the method.

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 chose `GET` here, because listing streams is intuitively a READ operation.

3. In the method's **Setup** pane, choose **AWS Service**.

- a. For **AWS Region**, choose a region (e.g., **us-east-1**).
- b. For **AWS Service**, choose **Kinesis**.
- c. Leave **AWS Subdomain** blank.
- d. For **HTTP method**, choose **POST**.

Note

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

- e. For **Action Type**, choose **Use action name**.
- f. For **Action**, type **ListStreams**.
- g. For **Execution role**, type the ARN for your execution role.
- h. Leave the default of **Passthrough** for **Content Handling**.
- i. Choose **Save** to finish the initial setup of the method.

/streams - GET - Setup



Choose the integration point for your new method.

Integration type Lambda Function ⓘ
 HTTP ⓘ
 Mock ⓘ
 AWS Service ⓘ

AWS Region

AWS Service

AWS Subdomain

HTTP method

Action Type Use action name
 Use path override

Action

Execution role ⓘ

Content Handling ⓘ

Save

4. Still in the **Integration Request** pane, expand the **HTTP Headers** section:

- a. Choose **Add header**.
- b. In the **Name** column, type **Content-Type**.
- c. In the **Mapped from** column, type '**application/x-amz-json-1.1**'.
- d. Choose the check mark icon to save the setting.

We used 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.

5. Expand the **Body Mapping Templates** section:

- a. Choose **Add mapping template**.
- b. For **Content-Type**, type **application/json**.
- c. Choose the check mark icon to save the **Content-Type** setting. Choose **Yes, secure this integration** in **Change passthrough behavior**.
- d. Type **{ }** in the template editor.
- e. Choose the **Save** button to save the mapping template.

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.

▼ HTTP Headers

Name	Mapped from <small>i</small>	Caching
Content-Type	'application/x-amz-json-1.1'	<input type="checkbox"/>  

+ Add header

▼ Body Mapping Templates

Request body passthrough When there are no templates defined (recommended) i
 When no template matches the request Content-Type header i
 Never i

Content-Type
application/json 

+ Add mapping template

application/json

Generate template: 

```
1  {}
```



Cancel **Save**

6. Test the GET method on the Streams resource to invoke the `ListStreams` action in Kinesis:

From the API Gateway console, select the `/streams/GET` entry from the **Resources** pane, choose the **Test** invocation option, and then choose **Test**.

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  
  
{  
    "ExclusiveStartShardId": "string",  
    "Limit": number,  
    "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 configure and test the GET method on a stream resource

1. Create a child resource with the `{stream-name}` path variable under the previously created `/streams` resource.

New Child Resource

Use this page to create a new child resource for your resource.

Configure as proxy resource i

Resource Name*

Resource Path* You can add path parameters using brackets. For example, the resource path `{username}` represents a path parameter called 'username'. Configuring `/streams/{proxy+}` as a proxy resource catches all requests to its sub-resources. For example, it works for a `GET` request to `/streams/foo`. To handle requests to `/streams`, add a new ANY method on the `/streams` resource.

Enable API Gateway CORS i

* Required

2. Add the `POST`, `GET`, and `DELETE` HTTP verbs to this resource.

After the methods are created on the resource, the structure of the API looks like the following:

Resources Actions ▾

-  /
-  /streams
 - GET
 - /{stream-name}
 - DELETE
 - POST
 - GET

3. Set up the `GET /streams/{stream-name}` method to call the `POST /?` `Action=DescribeStream` action in Kinesis, as shown in the following.

/streams/{stream-name} - GET - Setup



Choose the integration point for your new method.

Integration type Lambda Function i

HTTP i

Mock i

AWS Service i

AWS Region

AWS Service

AWS Subdomain

HTTP method

Action Type Use action name

Use path override

Action

Execution role i

Content Handling i

Save

4. Add the following Content-Type header mapping to the integration request:

```
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.

5. 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.

6. Test the `GET /stream/{stream-name}` method to invoke the `DescribeStream` action in Kinesis:

From the API Gateway console, select `/streams/{stream-name}/GET` in the **Resources** pane, choose **Test** to start testing, type the name of an existing Kinesis stream in the **Path** field for `stream-name`, and 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. Set up the `POST /streams/{stream-name}` method to call `POST /?Action=CreateStream` action in Kinesis. The task follows the same procedure to set up the `GET /streams/{stream-name}` method provided that you replace the `DescribeStream` action by `CreateStream`.
2. Add the following Content-Type header mapping to the integration request:

```
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.

3. 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.

4. Test the `POST /streams/{stream-name}` method to create a named stream in Kinesis:

From the API Gateway console, select `/streams/{stream-name}/POST` in the **Resources** pane, choose **Test** to start testing, type the name of an existing Kinesis stream in **Path** for `stream-name`, and 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. Set up the `DELETE /streams/{stream-name}` method to integrate with the `POST /?Action=DeleteStream` action in Kinesis. The task follows the same procedure to set up the `GET /streams/{stream-name}` method provided that you replace the `DescribeStream` action by `DeleteStream`.
2. Add the following Content-Type header mapping to the integration request:

```
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.

3. 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`.

4. Test the `DELETE` method to delete a named stream in Kinesis:

From the API Gateway console, select the `/streams/{stream-name}/DELETE` method node in the **Resources** pane, choose **Test** to start testing, type the name of an existing Kinesis stream in **Path** for `stream-name`, and 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
```

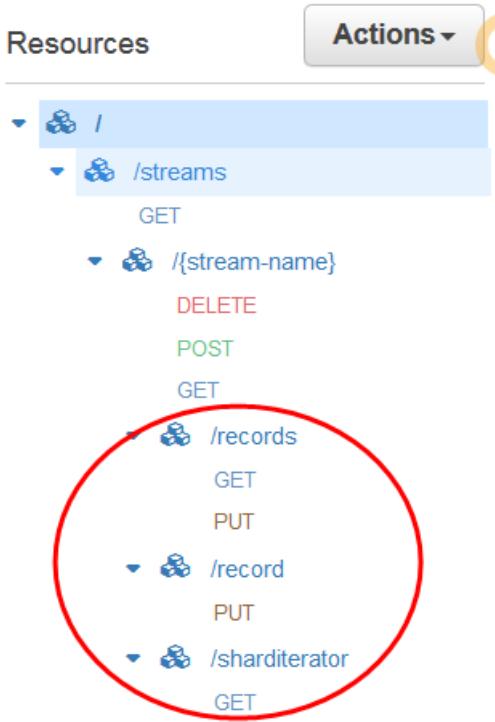
```
Content-Length: PayloadSizeBytes

{
    "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`).

The following figure shows the API structure of resources after the methods are created:



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. Set up the `PUT` method, as shown in the following:

[Method Execution](#)

/streams/{stream-name}/record - PUT - Integration Request



Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function

HTTP

Mock

AWS Service

AWS Region us-east-1

AWS Service Kinesis

AWS Subdomain

HTTP method POST

Action PutRecord

Execution role arn:aws:iam::█████████████████████:role/apigAwsProxyRole

Credentials cache Do not add caller credentials to cache key

Content Handling Passthrough

2. Add the following request parameter mapping to set the Content-Type header to an AWS-compliant version of JSON in the integration request:

```
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.

3. 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.

4. 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-00000000004"  
}
```

To set up and test the `PUT /streams/{stream-name}/records` method to invoke PutRecords in Kinesis

1. Set up the `PUT /streams/{stream-name}/records` method, as shown in the following:

[Method Execution](#)

/streams/{stream-name}/records - PUT - Integration Request



Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function

HTTP

Mock

AWS Service

AWS Region us-east-1

AWS Service Kinesis

AWS Subdomain

HTTP method POST

Action PutRecords

Execution role arn:aws:iam::7...:role/apigAwsProxyRole

Credentials cache Do not add caller credentials to cache key

Content Handling Passthrough

2. Add the following request parameter mapping to set the Content-Type header to an AWS-compliant version of JSON in the integration request:

```
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.

3. Add the following body 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.

4. 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. Set up integration for the GET /streams/{stream-name}/sharditerator method, as shown in the following:



Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function ⓘ
 HTTP ⓘ
 Mock ⓘ
 AWS Service ⓘ

AWS Region us-east-1 ✎

AWS Service Kinesis ✎

AWS Subdomain ✎

HTTP method POST ✎

Action GetShardIterator ✎

Execution role arn:aws:iam::7...:role/apigAwsProxyRole ✎

Credentials cache Do not add caller credentials to cache key ✎

Content Handling Passthrough ✎ ⓘ

2. 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:

[Method Execution](#)

/streams/{stream-name}/sharditerator - GET - Method Req...



Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization NONE

Request Validator NONE

API Key Required false

▶ Request Paths

▼ URL Query String Parameters

Name	Required	Caching	
shard-id	<input type="checkbox"/>	<input type="checkbox"/>	

[Add query string](#)

▶ HTTP Request Headers

▶ Request Body

▶ SDK Settings

In the following body-mapping template, we set the shard-id query parameter value to the ShardId property value of the JSON payload as the input to the GetShardIterator action in Kinesis.

3. Configure the body 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')"  
}
```

4. 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. Set up the GET /streams/{stream-name}/records method, as shown in the following:

[Method Execution](#)

/streams/{stream-name}/records - GET - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type Lambda Function [i](#)
 HTTP [i](#)
 Mock [i](#)
 AWS Service [i](#)

AWS Region us-east-1 [e](#)

AWS Service Kinesis [e](#)

AWS Subdomain [e](#)

HTTP method POST [e](#)

Action GetRecords [e](#)

Execution role arn:aws:iam::777777777777:role/apigAwsProxyRole [e](#)

Credentials cache Do not add caller credentials to cache key [e](#)

Content Handling Passthrough [e](#) [i](#)

2. 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, as shown in the following:

[Method Execution](#) /streams/{stream-name}/records - GET - Method Request 

Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization NONE  

Request Validator NONE  

API Key Required false 

▶ Request Paths

▶ URL Query String Parameters

▼ HTTP Request Headers

Name	Required	Caching	
Shard-Iterator	<input type="checkbox"/>	<input type="checkbox"/>	 

 [Add header](#)

▶ Request Body 

▶ SDK Settings

3. Set up the following 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')"  
}
```

4. Using the **Test** option in the API Gateway console, type 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": "AAAAAAAAAAFAF...",  
    "Records": [ ... ]  
}
```

Swagger Definitions of a Sample API as a Kinesis Proxy

The following shows the Swagger Definitions of the sample API as a Kinesis proxy used for this tutorial.

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-03-31T18:25:32Z",
    "title": "KinesisProxy"
  },
  "host": "wd4zclrobb.execute-api.us-east-1.amazonaws.com",
  "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": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n}"
          },
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
          "httpMethod": "POST",
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
          },
          "type": "aws"
        }
      }
    },
    "/streams/{stream-name)": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path"
          }
        ]
      }
    }
  }
}
```

```

        "in": "path",
        "required": true,
        "type": "string"
    }
],
"responses": {
    "200": {
        "description": "200 response",
        "schema": {
            "$ref": "#/definitions/Empty"
        }
    }
},
"x-amazon-apigateway-integration": {
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "responses": {
        "default": {
            "statusCode": "200"
        }
    },
    "requestTemplates": {
        "application/json": "{\n            \"StreamName\": \"$input.params('stream-\nname')\"\n        },\n        \"uri\": \"arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream\", \n        \"httpMethod\": \"POST\", \n        \"type\": \"aws\"\n    }
},
"post": {
    "consumes": [
        "application/json"
    ],
    "produces": [
        "application/json"
    ],
    "parameters": [
        {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
        }
    ],
    "responses": {
        "200": {
            "description": "200 response",
            "schema": {
                "$ref": "#/definitions/Empty"
            }
        }
    },
    "x-amazon-apigateway-integration": {
        "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
        "responses": {
            "default": {
                "statusCode": "200"
            }
        },
        "requestTemplates": {
            "application/json": "{\n                \"ShardCount\": 5,\n                \"StreamName\": \"$input.params('stream-name')\"\n            },\n            \"uri\": \"arn:aws:apigateway:us-east-1:kinesis:action/CreateStream\", \n            \"httpMethod\": \"POST\", \n            \"requestParameters\": {\n

```

```

        "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
    },
    "type": "aws"
}
},
"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": {
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "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": "$input.params('Content-Type')"
        }
    },
    "requestTemplates": {
        "application/json": "{\n            \"StreamName\": \"$input.params('stream-name')\"\n        }"
    },
    "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DeleteStream",
    "httpMethod": "POST",
    "requestParameters": {
        "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
    },
    "type": "aws"
}
},
"/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": {
                    "$ref": "#/definitions/Empty"
                }
            }
        }
    },
    "x-amazon-apigateway-integration": {
        "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
        "responses": {
            "default": {
                "statusCode": "200"
            }
        },
        "requestTemplates": {
            "application/json": "{\n                \"StreamName\": \"$input.params('stream-name')\"\n            }\n            \"Data\": \"$util.base64Encode($input.json('.Data'))\"\n            \"PartitionKey\": \"$input.path('.PartitionKey')\"\n        }"
        },
        "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecord",
        "httpMethod": "POST",
        "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
        },
        "type": "aws"
}
}
}

```

```

"/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": {
                    "$ref": "#/definitions/Empty"
                }
            }
        },
        "x-amazon-apigateway-integration": {
            "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
            "responses": {
                "default": {
                    "statusCode": "200"
                }
            },
            "requestTemplates": {
                "application/json": "{\n      \"ShardIterator\": \"$input.params('Shard-Iterator')\"\n    }"
            },
            "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetRecords",
            "httpMethod": "POST",
            "requestParameters": {
                "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
            },
            "type": "aws"
        }
    },
    "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": "path",
        "required": true,
        "type": "string"
    },
    {
        "in": "body",
        "name": "PutRecordsMethodRequestPayload",
        "required": true,
        "schema": {
            "$ref": "#/definitions/PutRecordsMethodRequestPayload"
        }
    }
],
"responses": {
    "200": {
        "description": "200 response",
        "schema": {
            "$ref": "#/definitions/Empty"
        }
    }
},
"x-amazon-apigateway-integration": {
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "responses": {
        "default": {
            "statusCode": "200"
        }
    },
    "requestTemplates": {
        "application/json": "{$input.params('stream-name')}\n"
        "Records": "[\n            #foreach($elem in $input.path('$records'))\n                {\n                    \"Data\": \"$util.base64Encode($elem.data)\",\n                    \"PartitionKey\": \"$elem.partition-key\"\n                }#if($foreach.hasNext),#end\n            #end\n        ]\n    }",
        "application/x-amz-json-1.1": "#set($inputRoot = $input.path('$'))\n{\n    \"StreamName\": \"$input.params('stream-name')\", \n    \"records\" : [\n        #foreach($elem in $inputRoot.records)\n            {\n                \"Data\" : \"$elem.data\", \n                \"PartitionKey\" : \"$elem.partition-key\"\n            }#if($foreach.hasNext),#end\n        #end\n    ]\n}"
    },
    "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecords",
    "httpMethod": "POST",
    "requestParameters": {
        "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
    },
    "type": "aws"
}
},
"/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": {
            "$ref": "#/definitions/Empty"
        }
    }
},
"x-amazon-apigateway-integration": {
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "responses": {
        "default": {
            "statusCode": "200"
        }
    },
    "requestTemplates": {
        "application/json": "{\n            \"ShardId\": \"$input.params('shard-id')\", \n            \"ShardIteratorType\": \"TRIM_HORIZON\", \n            \"StreamName\": \"$input.params('stream-name')\"\n        }"
    },
    "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetShardIterator",
    "httpMethod": "POST",
    "requestParameters": {
        "integration.request.header.Content-Type": "'application/x-amz-json-1.1'"
    },
    "type": "aws"
}
},
"definitions": {
    "PutRecordsMethodRequestPayload": {
        "type": "object",
        "properties": {
            "records": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "data": {
                            "type": "string"
                        },
                        "partition-key": {
                            "type": "string"
                        }
                    }
                }
            }
        }
    },
    "Empty": {
        "type": "object"
    }
}
}

```

Amazon API Gateway REST API

When you use the Amazon API Gateway console to create, configure, update, and deploy an API, the console calls the API Gateway REST API behind the scenes to make things happen.

When you use AWS Command Line Interface to create, configure, update, and deploy an API, the AWS CLI tool calls the API Gateway REST API as well. For an example, see [Create an API using API Gateway and Test It](#) in the *AWS Lambda Developer Guide*. For more information, see [AWS Command Line Interface User Guide](#).

When you use an [AWS SDK](#) to create, configure, update, and deploy an API, the SDK calls the API Gateway REST API behind the scenes.

Instead, you can call the API Gateway REST API directly to create, configure, update, and deploy an API in API Gateway.

For more information on how to use the API Gateway REST API, see [Amazon API Gateway REST API Reference](#).

Amazon API Gateway Limits, Pricing and Known Issues

Topics

- [API Gateway Limits \(p. 582\)](#)
- [API Gateway Pricing \(p. 584\)](#)
- [Known Issues \(p. 585\)](#)

API Gateway Limits

Unless noted otherwise, the limits can be increased upon request. To request a limit increase, contact the [AWS Support Center](#).

When authorization is enabled on a method, the maximum length of the method's ARN (e.g., `arn:aws:execute-api:{region-id}:{account-id}:{api-id}/{stage-id}/{method}/{resource}/{path}`) 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.

Header values are limited to 10240 bytes.

API Gateway Limits for Configuring and Running an API

The following limits apply to configuring and running an API in Amazon API Gateway.

Resource or Operation	Default Limit	Can Be Increased
Throttle limits per account per region	10000 request per second (rps) with an additional burst capacity provided by the token bucket algorithm , using a maximum bucket capacity of 5000 requests. Note The burst limit cannot be increased.	Yes
APIs (or RestApis) per account per region	60	Yes
Maximum length, in characters, of resource policy	8092	Yes
API keys per account per region	500	Yes
Client certificates per account per region	60	Yes

Resource or Operation	Default Limit	Can Be Increased
Custom authorizers per API	10	Yes
Documentation parts per API	2000	Yes
Resources per API	300	Yes
Stages per API	10	Yes
Usage plans per account per region	300	Yes
Usage plans per API key	10	Yes
VPC links per account per region	5	Yes
API caching TTL	300 seconds by default and configurable between 0 and 3600 by an API owner.	Not for the upper bound (3600)
Integration timeout	50 milliseconds - 29 seconds for all integration types, including Lambda, Lambda proxy, HTTP, HTTP proxy, and AWS integrations.	Not for the lower or upper bounds.
Header value size	10240 Bytes	No
Payload size	10 MB	No
Tags per stage	50	No
Number of iterations in a <code>#foreach ... #end</code> loop in mapping templates	1000	No
ARN length of a method with authorization	1600 bytes	No

For `restapi:import` or `restapi:put`, the maximum size of the API definition file is 2MB.

All of the per API limits can only be increased on specific APIs.

API Gateway Limits for Creating, Deploying and Managing an API

The following fixed limits 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 limits cannot be increased.

Action	Default Limit	Can Be Increased
CreateRestApi	2 requests per minute (rpm) per account.	No
ImportRestApi	2 requests per minute per account	No
PutRestApi	60 requests per minute per account	No
DeleteRestApi	2 requests per minute per account	No
CreateDeployment	3 requests per minute per account	No
UpdateAccount	3 requests per minute per account	No
GetResources	150 requests per minute per account	No
CreateResource	300 requests per minute per account	No
DeleteResource	300 requests per minute per account	No
CreateDomainName	2 requests per minute per account	No
UpdateUsagePlan	3 requests per minute per account	No
Other operations	No limit up to the total account limit.	No
Total operations	10 request per second (rps) with a burst limit of 40 requests.	No

API Gateway Pricing

For general API Gateway region-specific pricing information, see [Amazon API Gateway Pricing](#).

:

The following lists the exceptions of the general pricing scheme:

- API caching in Amazon API Gateway is not eligible for the AWS Free Tier.
- Calling methods with the [authorization type](#) of `AWS_IAM`, `CUSTOM`, and `COGNITO_USER_POOLS` are not charged for authorization and authentication failures.
- Calling methods requiring API keys are not charged when API keys are missing or invalid.
- API Gateway-throttled requests are not charged when the request rate or burst exceed the pre-configured limits.
- Usage plan-throttled requests are not charged when rate limits or quota exceed the pre-configured limits.

Known Issues

- API Gateway does not support wild-card subdomain name (of the * .domain form). However, it supports wild-card certificates, namely, a certificate for a wild-card subdomain name.
- 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 plain text pipe character (|) is not supported for any request URL query string and must be URL-encoded.
- Paths of /ping and /sping 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.
- 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.
- 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.
- The following backends may not support SSL client authentication in a compatible way with API Gateway:
 - [NGINX](#)
 - [Heroku](#)
- API Gateway supports most of the [Swagger specification](#), with the following exceptions:
 - API Gateway models are defined using [JSON schema draft 4](#), instead of the JSON schema used by Swagger.
 - The additionalProperties field is not supported in Models.
 - The allOf field is not supported in Models.
 - 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 by API Gateway
 - pattern is not supported by API Gateway
 - The readOnly field is not supported.
 - Response definitions of the "500": {"\$ref": "#/responses/UnexpectedError"} form is not supported in the Swagger document root. To work around this, replace the reference by the inline schema.
 - Numbers of the Int32 or Int64 type is 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 enacts the following restrictions and limitations when handling methods with either Lambda integration or HTTP integration.
 - Duplicated query string parameters are not supported.
 - Duplicated headers are not supported.
 - The Host header will not be forwarded to HTTP endpoints.
- The following headers may be remapped to x-amzn-Remapped-**HEADER** when sent to your integration endpoint or sent back by your integration endpoint:
 - Accept
 - Accept-Charset
 - Accept-Encoding
 - Age
 - Authorization
 - Connection
 - Content-Encoding
 - Content-Length
 - Content-MD5
 - Content-Type
 - Date
 - Expect
 - Host
 - Max-Forwards
 - Pragma
 - Proxy-Authenticate
 - Range
 - Referer
 - Server
 - TE
 - Trailer
 - Transfer-Encoding
 - Upgrade
 - User-Agent
 - Via
 - Warn
 - WWW-Authenticate
- 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.

Document History

The following table describes the important changes to the documentation since the last release of the API Gateway Developer Guide.

- **Latest documentation update:** December 19, 2017

Change	Description	Date Changed
Cross Account Lambda Authorizers and Integrations	Use an AWS Lambda function from a different AWS account as a Lambda authorizer function or as an API integration backend. The other account can be in any region where Amazon API Gateway is available. For more information, see the section called "Configure Cross-Account Lambda Authorizer" (p. 285) and the section called "Build an API with Cross-Account Lambda Proxy Integration" (p. 27) .	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" (p. 245) .	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 for an API Stage" (p. 395) .	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 "Enable Payload Compression" (p. 217) .	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 require the key. For more information, see the section called "Expose API Key Source" (p. 315) .	December 19, 2017
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 Pools" (p. 286) .	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 "Build an API with Private Integration" (p. 74) and the section called "Set up Private Integrations" (p. 146) .	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	November 28, 2017

Change	Description	Date Changed
	execution and access logged in separate CloudWatch Logs logs. For more information, see the section called "Set up a Canary Release Deployment" (p. 397) .	
Access Logging	Log client access to your API with data derived from \$context variables (p. 191) in a format of your choosing. For more information, see the section called "Set up API Logging" (p. 382) .	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" (p. 419) and the section called "Use a Ruby SDK Generated by API Gateway" (p. 466) .	November 20, 2017
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" (p. 103) and the section called "Set up a Regional Custom Domain Name" (p. 442) .	November 2, 2017
Custom request authorizer	Use custom request authorizer to supply user-authenticating 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 (p. 272) .	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 Set up Gateway Responses to Customize Error Responses (p. 155) .	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 <code>integration.response.body</code> 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 (p. 141) .	June 6, 2017
Throttling limits increase	Increase the account-level steady-state request rate limit to 10,000 requests per second (rps) and the burst limit to 5000 concurrent requests. For more information, see Throttle API Requests for Better Throughput (p. 376) .	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 parameters are set and not blank, and verifies that the format of applicable payloads conforms to the configured model. For more information, see Enable Request Validation in API Gateway (p. 221) .	April 11, 2017

Change	Description	Date Changed
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 certificate into ACM. You then refer to the certificate's ARN when setting a custom domain name for your APIs. For more information, see Set up Custom Domain Name for an API in API Gateway (p. 432) .	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 (p. 460) .	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 API as SaaS (p. 453) .	December 1, 2016
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 an API Gateway API (p. 329) .	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 (p. 279) .	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-encoded string, or as a passthrough without modifications. For more information, see Support Binary Payloads in API Gateway (p. 199) .	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 (p. 122) .	September 20, 2016
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-upon request rates and quotas. For more information, see Create and Use API Gateway Usage Plans (p. 314) .	August 11, 2016

Change	Description	Date Changed
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 Use Amazon Cognito User Pools (p. 286) .	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 (p. 480) .	July 28, 2016
Enabling certificate 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 (p. 441) .	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 Build an API Gateway API from an Example (p. 9) and Build an API with HTTP Custom Integration (p. 44) .	April 5, 2016
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 Import an API into API Gateway (p. 238) .	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 Reference (p. 191) .	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 (p. 380) and Invalidate an API Gateway Cache Entry (p. 380) . Improve the console experience for managing API request throttling. For more information, see Throttle API Requests for Better Throughput (p. 376) .	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 (p. 272) .	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 Import an API into API Gateway (p. 238) and Export an API (p. 414) .	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 (p. 187) .	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 Set up Stage Variable for API Deployment (p. 385) .	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 Enable CORS for a Resource (p. 267) .	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 Use Client-Side SSL Certificates for Authentication by the Backend (p. 293) .	September 22, 2015
Mock integration of methods	Learn how to mock-integrate an API with Amazon API Gateway (p. 152) . This feature enables developers to generate API responses from API Gateway directly without the need for a final integration backend beforehand.	September 1, 2015
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 Reference (p. 191) .	August 28, 2015
Swagger integration	Use the Swagger import tool on GitHub to import Swagger API definitions into Amazon API Gateway. Learn more about API Gateway Extensions to Swagger (p. 486) 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 Reference (p. 191) .	July 18, 2015
Initial public release	This is the initial public release of the <i>API Gateway Developer Guide</i> .	July 9, 2015

AWS Glossary

For the latest AWS terminology, see the [AWS Glossary](#) in the *AWS General Reference*.