Amazon API Gateway

Developer Guide

Amazon API Gateway: Developer Guide

Table of Contents

What Is Amazon API Gateway?

Amazon API Gateway consists of two services: the API Gateway control service and API Gateway execution service. The control service lets you create a RESTful API to expose selected back-end features. The back end can be another AWS service, such as AWS Lambda or Amazon DynamoDB, or it can be an existing web application. The execution service lets an app call the API to access the exposed back-end features. The app can interact with the API using standard HTTP protocols or using a platform- or language-specific SDK generated by the API creator.

The API you create in API Gateway consists of a set of resources and methods. A resource is a logical entity that can be accessed through a resource path using the API. A resource can have one or more operations that are defined by appropriate HTTP verbs such as GET, POST, and DELETE. A combination of a resource path and an operation identify a method in the API. Each method corresponds to a REST API request submitted by the user of your API and the corresponding response returned to the user. API Gateway integrates the method with a targeted back end by mapping the method request to an integration request acceptable by the back end and then mapping the integration response from the back end to the method response returned to the user. As an API developer, you can configure how methods are mapped to integrations and vice versa by stipulating what parameters to use and specifying mapping templates to transform payloads of given data models.

You can create an API by using the API Gateway management console, described in Getting Started (p. 4), or by using the API Gateway API Gateway REST API (p. 278). In addition, you can integrate API creation with AWS CloudFormation templates or API Gateway Extensions to Swagger (p. 115). For a list of regions where API Gateway is available, as well as the associated control service endpoints, see Regions and Endpoints.

API Gateway helps developers deliver robust, secure, and scalable mobile and web application back ends. API Gateway allows developers 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, developers can create and operate APIs for their back-end services without developing and maintaining 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 back-end 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 either be provided by a publicly accessible endpoint that API Gateway proxies call, or it can be entirely run as a Lambda function.

To better understand the terminology used in this documentation, you may find it useful to peruse the API Gateway Concepts (p. 2) section.

Amazon API Gateway Concepts

API Gateway	An AWS service that 1) supports creating, deploying and managing a RESTful application programming interface (API) to expose back-end HTTP endpoints, AWS Lambda function, or other AWS services; and 2) invoking exposed API methods through the front-end HTTP endpoints.
API Gateway API	A collection of resources and methods that are integrated with back-end HTTP endpoints, Lambda functions or other AWS services and can be deployed in one or more stages. API methods are invoked through front-end HTTP endpoints that can be associated with a registered custom domain names. Permissions to invoke a method can be granted using IAM roles and policies or API Gateway custom authorizers. An API can present a certificate to be authenticated by the back end. 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 who 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.
App user, end user or client endpoint	An entity that uses the application built by an app developer that interacts with APIs in Amazon API Gateway. An app user can be represented by an Amazon Cognito identity or a bearer token.
API Key	An alphanumeric string, which can be generated by API Gateway on behalf of an API owner or imported from an external source such as a CSV file, is used to identify an app developer of the API. An API owner can use API keys to permit or deny access of given 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 invocation by a client, it must be associated with one or more stages. A stage is a logical reference to a life-cycle status of your API (e.g., '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 back end 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 back end.
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 back end 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.
Mapping template	Scripts, expressed in Velocity Template Language (VTL), to transform a request body from the front-end data format to the back-end data format or to transform a response body from the back-end data format to the front-end data format. Mapping templates are specified in the integration request or integration response and they can reference data made available at run time in the forms of context and stage variables. An identity transformation is referred to as pass-through in which a payload is passed as-is from the client to the back end for a request and from the back end to the client for a response.
Model	Data schema specifying the data structure of a request or response payload. It is required for generating strongly typed SDK of an API, used for validating payload, and 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 with configurable throttling and quota limits enforced on individual client API keys.

Getting Started with Amazon API Gateway

The following walkthroughs include hands-on exercises, using the API Gateway console, to help you learn about API Gateway.

Topics

Get Ready to Use Amazon API Gateway

Before using API Gateway for the first time, you must have an AWS account set up. To create, configure and deploy an API in API Gateway, you must have appropriate IAM policy provisioned with permissible access rights to the API Gateway control service. To permit your API clients to invoke your API in API Gateway, you must set up the right IAM policy to allow the clients to call the API Gateway execution service. To allow API Gateway to invoke an AWS service in the back end, API Gateway must have permissions to assume the roles required to call the back-end AWS service. When an API Gateway API is set up to use AWS IAM roles and policies to control client access, the client must sign API Gateway API requests with Signature Version 4.

Understanding of these topics are important to use API Gateway and to follow the tutorials and instructions presented here. This section provides brief discussions of or quick references to these topics.

Topics

Sign Up for AWS

Go to http://aws.amazon.com/, choose **Create an AWS Account**, and follow the instructions therein.

Create an IAM User, Group or Role in Your AWS Account

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

To manage access for a user, you can create an IAM user, 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, you can 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 service, you can 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, you need to specify the ARN of an IAM role to control access the API's methods. Make sure that this is ready when creating an API.

Grant IAM Users Permissions to Access API Gateway Control and Execution Services

In AWS, access permissions are stated as policies. A policy created by AWS is a managed policy and one created by a user is an inline policy.

For the API Gateway control service, the managed policy of **AmazonAPIGatewayAdministrator** (`arn:aws:iam::aws:policy/AmazonAPIGatewayAdministrator`) grants the 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 stated permissions to a user, attach the policy to the user, a group containing the user. To attach a policy, see Attaching Managed Policies.

Attaching the preceding policy to an IAM user provides the user with access to all API Gateway control service actions and resources associated with the AWS account. To learn how to restrict IAM users to a limited set of API Gateway control service actions and resources, see Set IAM Permissions (p. 188).

For the API Gateway execution service, the managed policy of **AmazonAPIGatewayInvokeFullAccess** (`arn:aws:iam::aws:policy/AmazonAPIGatewayInvokeFullAccess`) provides full access to invoke an API in API Gateway:

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

Attaching the preceding policy to an IAM user provides the user with access to all API Gateway execution service actions and resources associated with the AWS account. To learn how to restrict IAM users to a limited set of API Gateway execution service actions and resources, see Set IAM Permissions (p. 188).

To grant the state permissions to a user, attach the policy to the user, a group containing the user. To attach a policy, see Attaching Managed Policies.

In this documentation, we will use managed policies, whenever possible. To create and use inline policies, see Working with Inline Policies.

> **Note**
> To complete the steps above, you must have permission to create the IAM policy and attach it to the desired IAM user.

Next Step

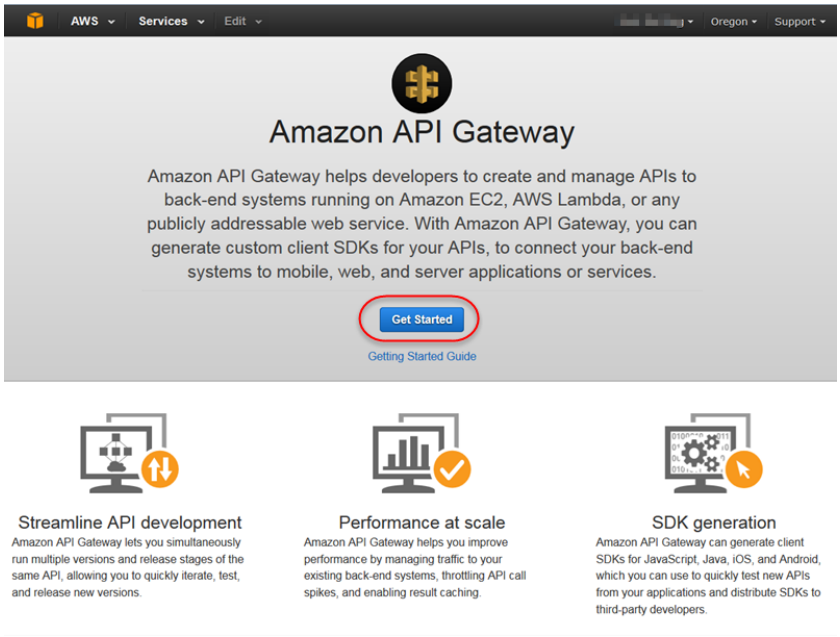
You are now ready to start using API Gateway. See Build and Test an API Gateway API from an Example (p. 6).

Build and Test an API Gateway API from an Example

The Amazon API Gateway console now provides an option for you to create an API Gateway API by example, with helpful hints provided along the way. If you are new to API Gateway, you may find it useful as a learning tool. The following steps walk you through using this create-by-example option to create and test the example API.

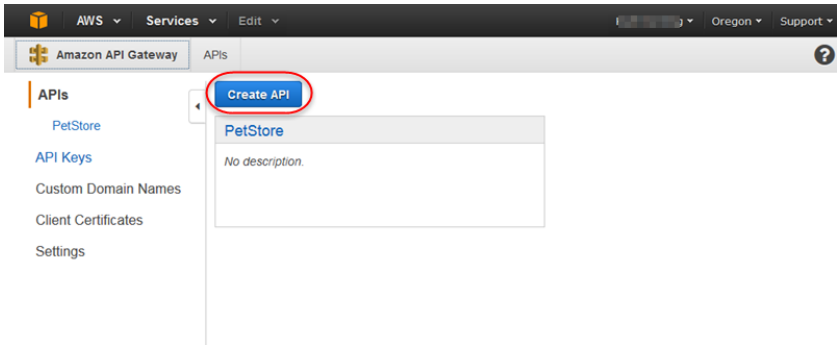
1. Do one of the following:

   a. For the first API in your account, choose **Get Started** from the API Gateway console welcome page:

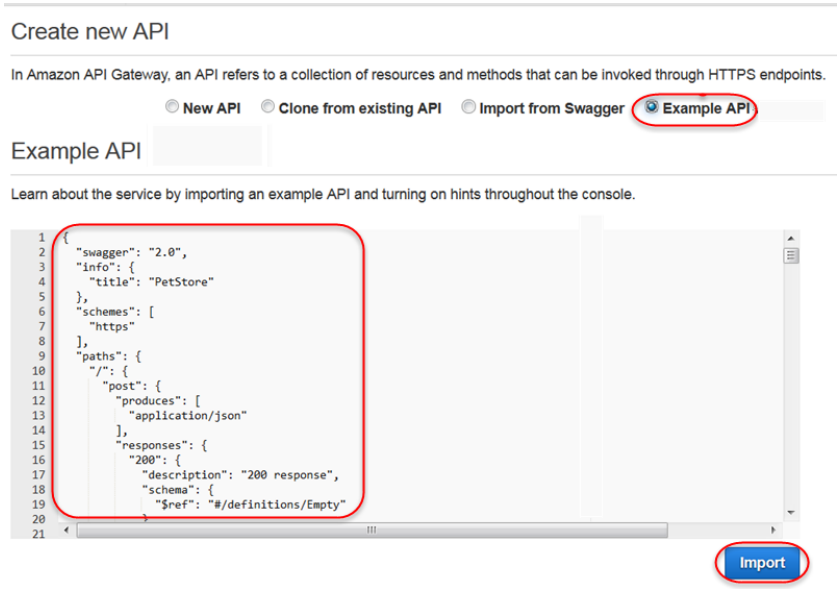


If prompted with a modal dialog box containing hints at a stage of the process, choose **OK** to close the modal dialog and continue.

b. For your next API, choose **Create API** from the API Gateway **APIs** home page:

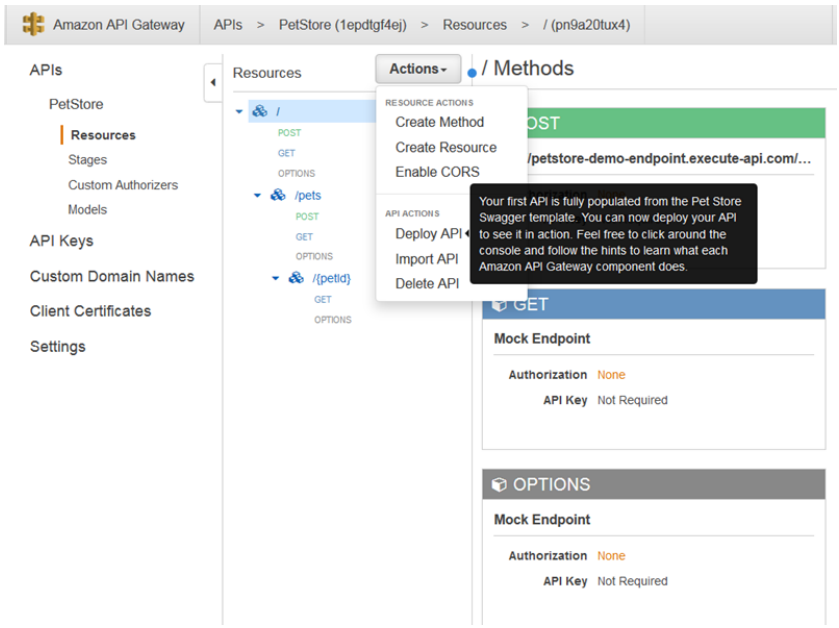


2. Under **Create new API**, select **Examples API** and then choose **Import** to create the example API. For your first API, the API Gateway console will start with this option as default.



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

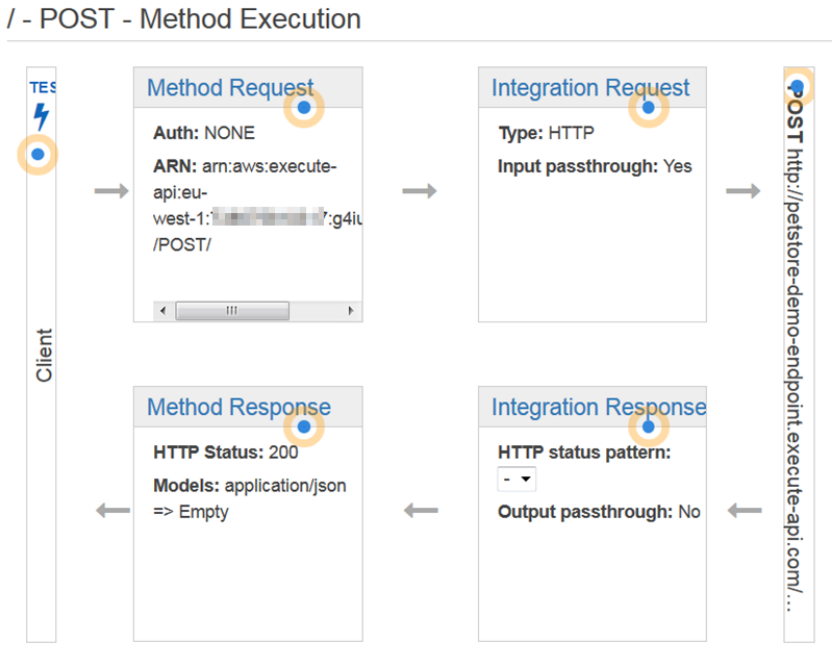
3. The resulting display shows the newly created API:



The API Gateway navigation pane on the left shows your available APIs, any API keys, custom domain names and client certificates that you created for your APIs, as well as the settings for logging your APIs' performance metrics. API-specific resources, deployment, custom authorizers and payload-mapping data models are organized under individual APIs.

The **Resources** pane in the middle shows the structure of the selected 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.

4. 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.

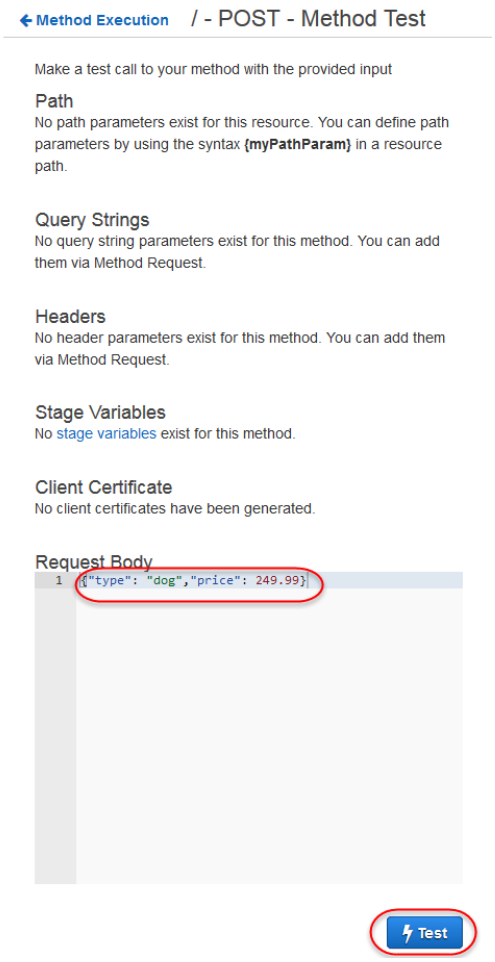


The resulting **Method Execution** pane for the chosen method presents a logical view of the method's structure and behaviors: a client accesses a back-end service by interacting with the API through **Method Request**. API Gateway translates the client request, if necessary, into the form acceptable to the back end before forwarding the request to the back end. The transformed request is known as the integration request and is depicted by **Integration Request** in the display. Similarly, the response from the back end goes through **Integration Response** and then **Method Response** before being received by the client. Again, if necessary, API Gateway maps the response from the form shaped in the back end to a form expected by the client.

For the `POST` method on this API's root (`/`) resource, the method's integration request shows that the method is integrated with the endpoint of `http://petstore-demo-endpoint.execute-api.com/petstore/pets` in the back end. The method request payload will be passed through to the integration request without modification. The `GET /` method request uses the `MOCK` integration type and is not tied to any endpoint in the back end. When the method is called, the API Gateway simply accepts the request and immediately returns a response, by way of from **Integration Response** to **Method Response**. You can use the mock integration to test an API without requiring a back-end endpoint. You can also use it to serve a local response. In fact, the example API uses it to return a local HTML page as the home page of the API. It uses a mapping template to generate the home page in **Integration Response**.

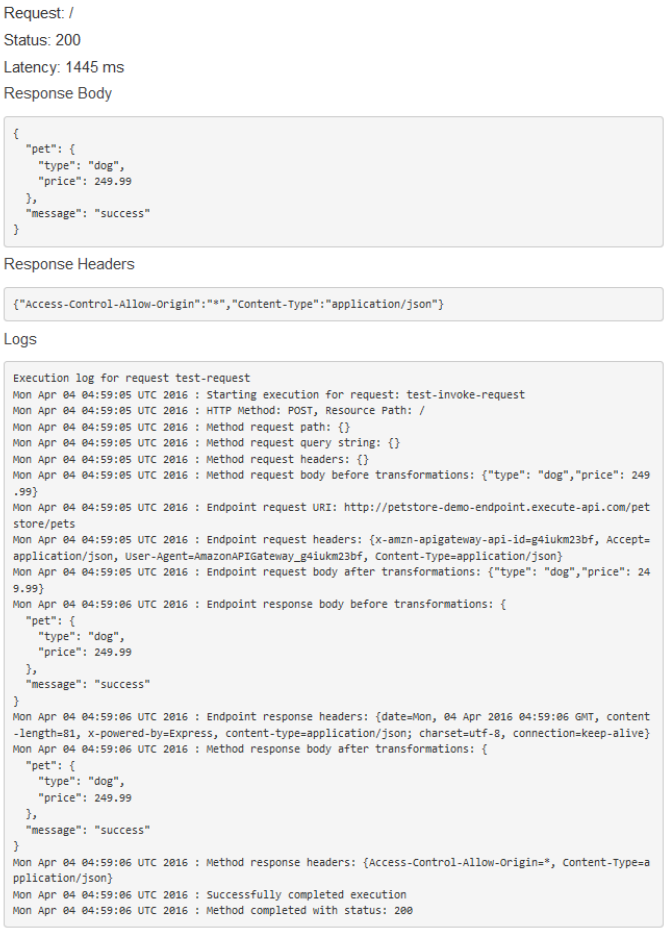
As an API developer, you control the behaviors of your API's front-end interactions by configuring the method request and a method response. You control the behaviors of your API's back-end interactions by setting up the integration request and integration response. They involve data mappings between a method and its corresponding integration. We will cover the method setup in Build an API Gateway API Step by Step (p. 14). For now, we focus on testing the API to provide an end-to-end user experience.

5. Choose **Test** shown on **Client** (as shown in the previous image) to start testing. Enter the following `{"type": "dog","price": 249.99}` payload into the **Request Body** before choosing the **Test** button.



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

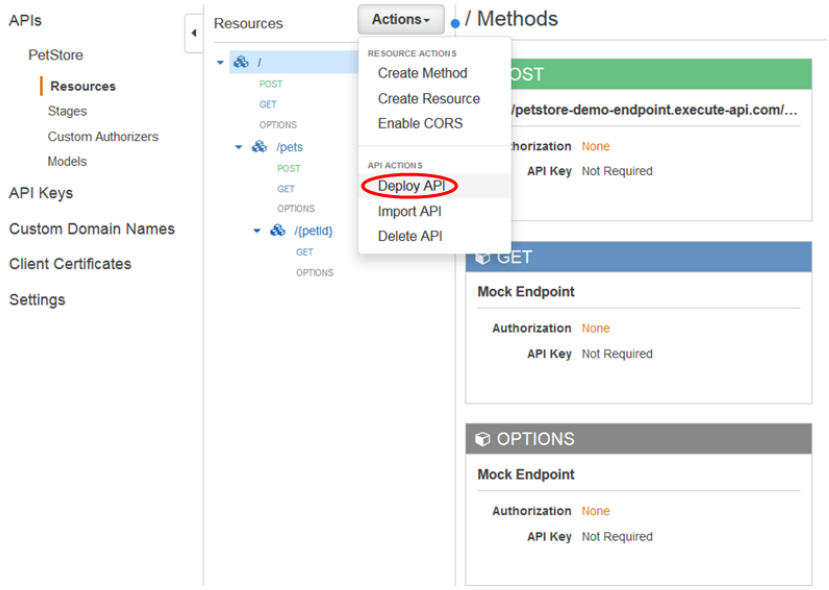
6. The results display as follows:



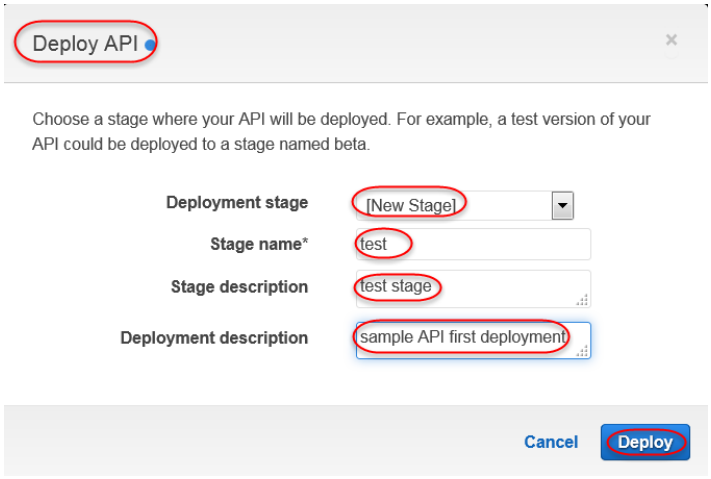
The **Logs** entry of the output shows the state changes from the method request to the integration request and from the integration response to the method response. This can be useful for troubleshooting any mapping errors that cause the request to fail. In this example, no mapping is applied: the method request is identical to the integration request and the integration response is the same as 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.

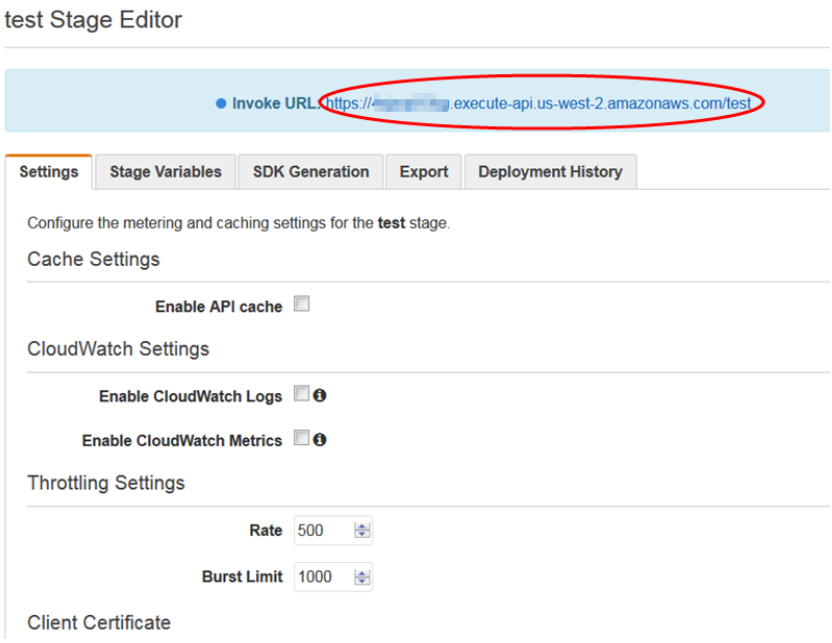
7. To deploy the sample API, select the **PetStore** API and the root ***/*** resource, and then choose **Deploy API** from the **Actions** menu.



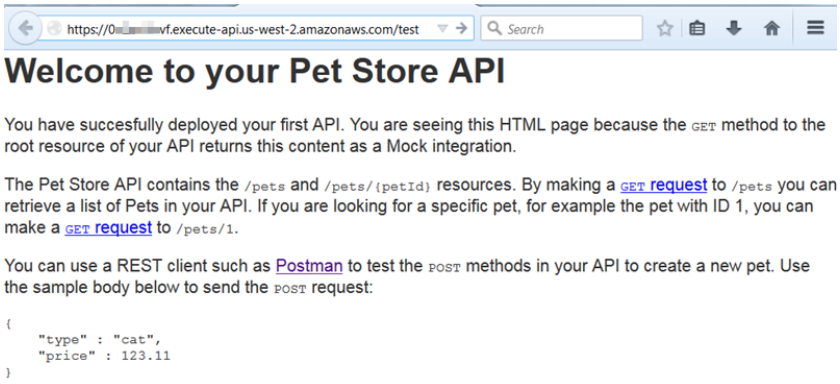
In **Deploy API**, for **Deployment stage**, choose **[New Stage]** because this is the first deployment of the API. Type a name (e.g., `test`) in **Stage name** and, optionally, type descriptions in **Stage description** and **Deployment description**. Choose **Deploy**.



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



8. On **Stage Editor**, follow the **Invoke URL** link to submit the `GET /` method request in a browser. The result, generated from the mapping template in the integration response, is shown as follows:



9. 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
}
```

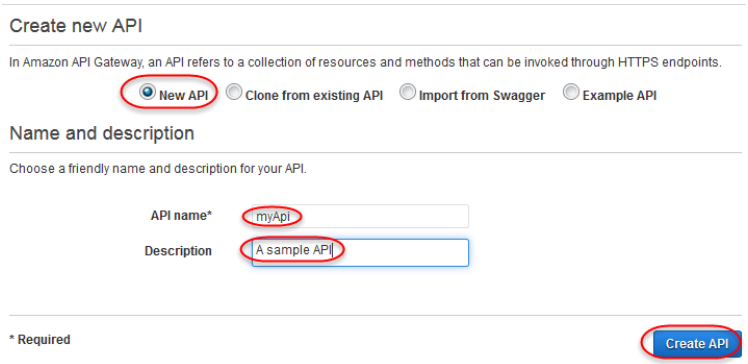
Invoking the API method as shown is possible because its **Authorization** type is set to `NONE`. If the `AWS_IAM` authorization were used, you would sign the request using the Signature Version 4 protocols. For an example of such a request, see Build an API Gateway API Step by Step (p. 14).

Build an API Gateway API Step by Step

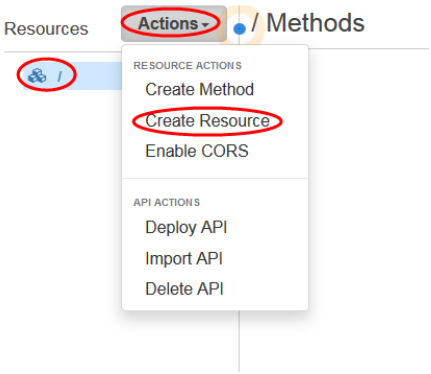
You can create an API in the Amazon API Gateway console from the ground up. In essence, you use the console as an API design studio to scope the API features, to experiment with its behaviors, to build the API, and to deploy your API in stages.

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.

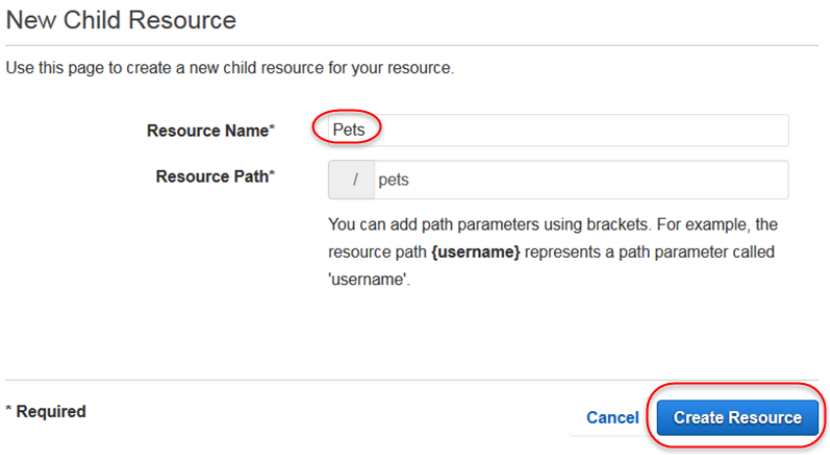
1. From **Create new API**, select **New API**, type a name in **API Name**, optionally add a description in **Description**, and then choose **Create API**.



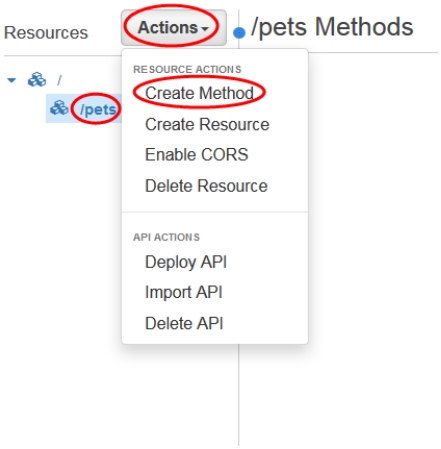
   As a result, an empty API is created. The **Resources** tree shows the root resource (`/`) without any methods. In this exercise, we will build the API as an HTTP proxy of the PetStore demo website (http://petstore-demo-endpoint.execute-api.come.) For illustration purposes, we will create a `/pets` resource 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**.



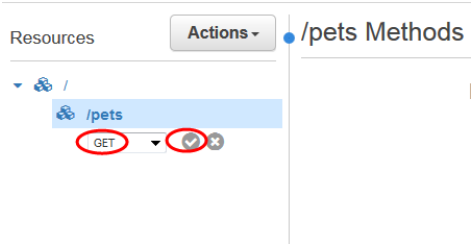
   Type `Pets` in **Resource Name**, leave the **Resource Path** value as given, and choose **Create Resource**.



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 checkmark icon to finish creating the method.



Note
Other options for an API method include:

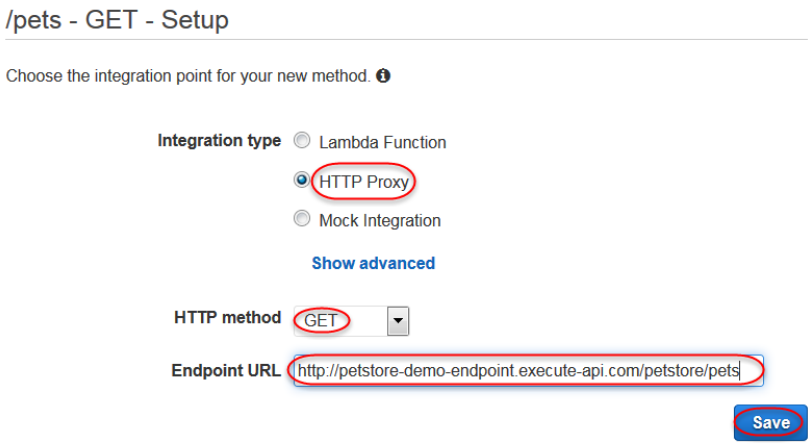
- **POST**, primarily used to create child resources.
- **PUT**, primarily used to update existing resources (and, although not recommended, can be used to create child resources).
- **DELETE**, used to delete resources.
- **PATCH**, used to update resources.

- **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 back end. The next step sets this up.

4. In the method's **Setup** pane, select **HTTP Proxy** 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, and then choose **Save**.

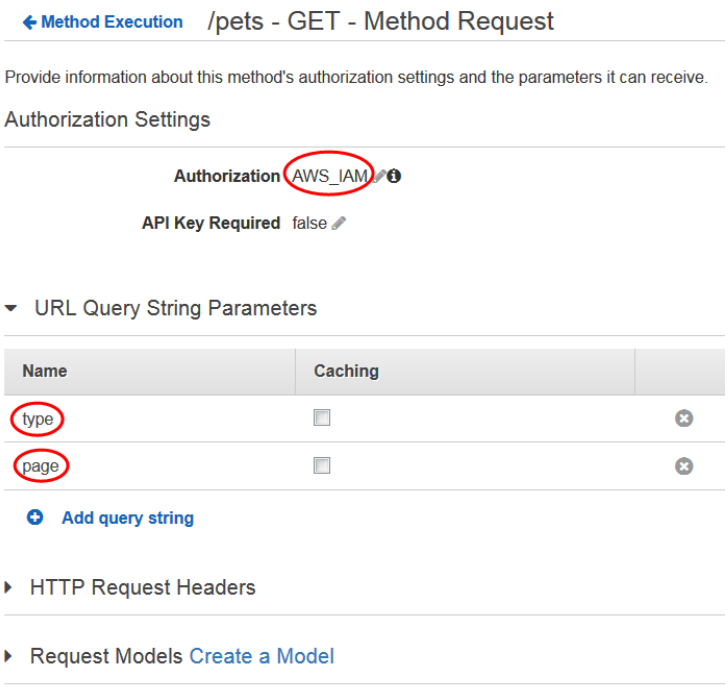
   Note
   For the integration request's **HTTP method**, you must choose one supported by the back end. For `HTTP Proxy` or `Mock integration`, it makes sense that the method request and the integration request use the same HTTP verb. For other integration types the method request will likely use an HTTP 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.



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.

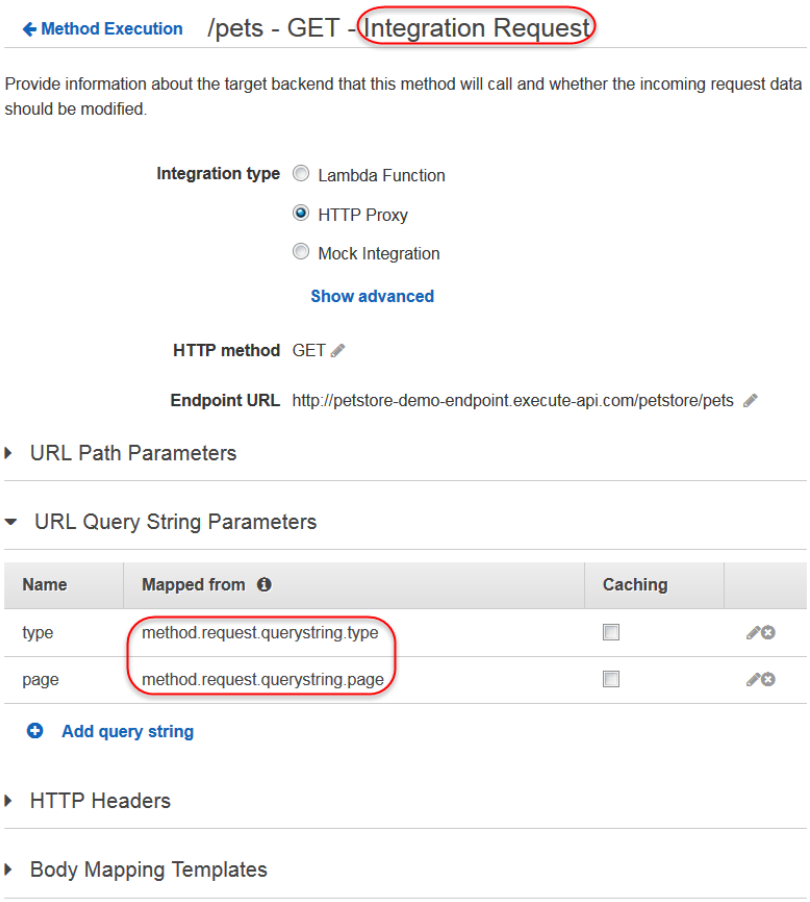
The PetStore website allows you to retrieve a list of `Pet` items by the pet type (e.g., "Dog" or "Cat") on a given page. It uses the `type` and `page` query string parameters to accept such input. As such, we must add the query string parameters to the method request and map them into the corresponding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



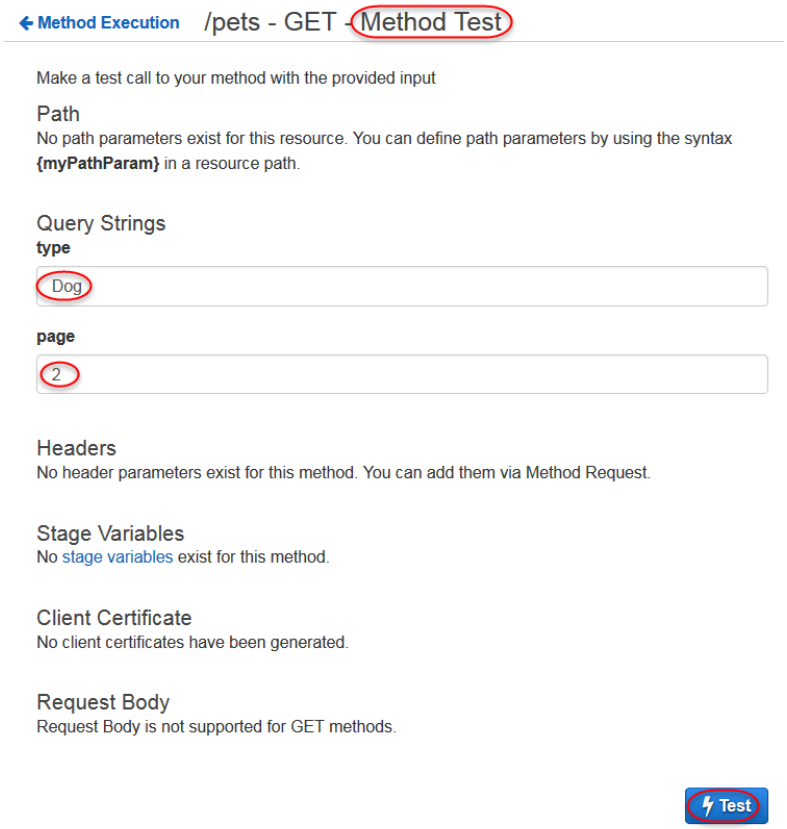
The client can now supply a pet type and a page number as query string parameters when submitting a request. These input parameters must be mapped into the integration's query string parameters to forward the input values to our PetStore website in the back end. Because the method uses `AWS_IAM`, you 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.



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** from the GET-on-Pets method's **Method Execution** pane. In the **Method Test** pane, enter **Dog** and **2** for the **type** and **page** query strings, respectively, and then choose **Test**.



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

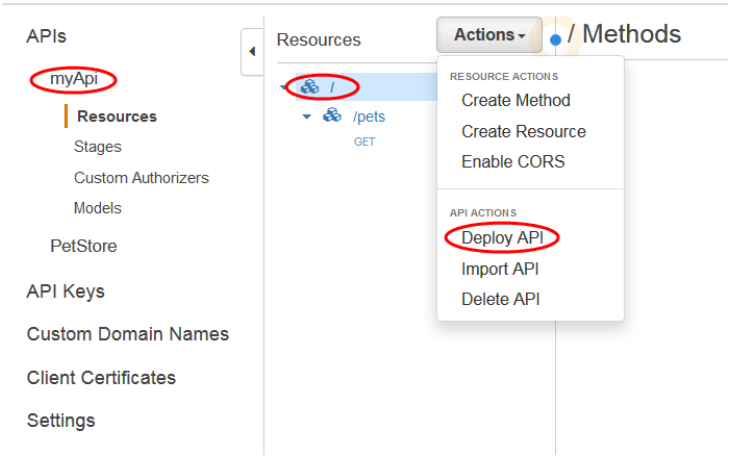

```
{"Content-Type":"application/json"}
```


Logs

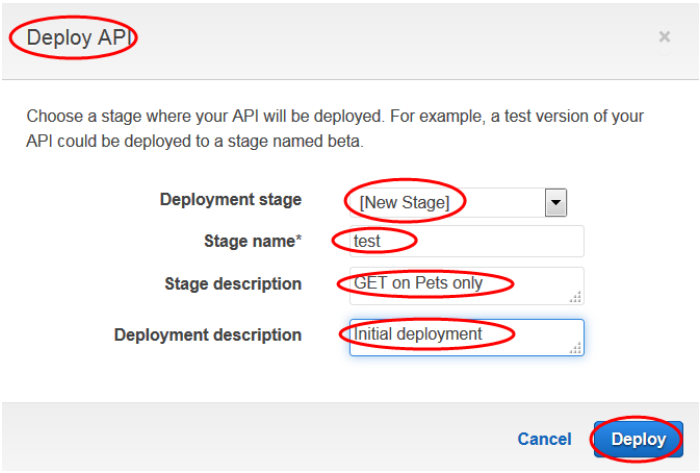
```
Execution log for request test-request
Mon Apr 04 05:48:01 UTC 2016 : Starting execution for request: test-invoke-request
Mon Apr 04 05:48:01 UTC 2016 : HTTP Method: GET, Resource Path: /pets
Mon Apr 04 05:48:01 UTC 2016 : Method request path: {}
Mon Apr 04 05:48:01 UTC 2016 : Method request query 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.

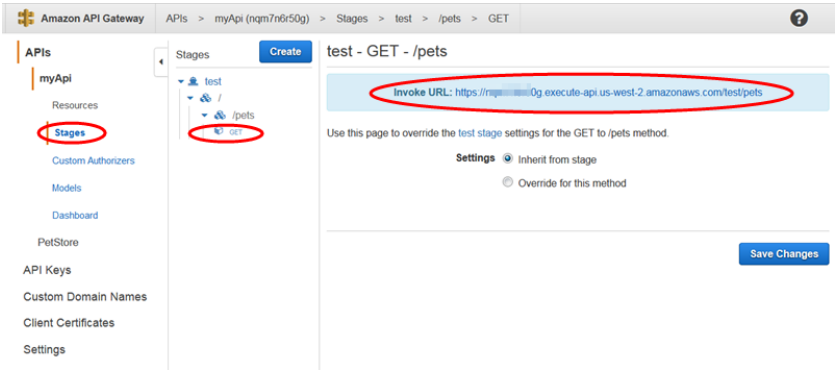
8. To deploy the API, select the API and then choose **Deploy API** from the **Actions** drop-down menu.



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**.



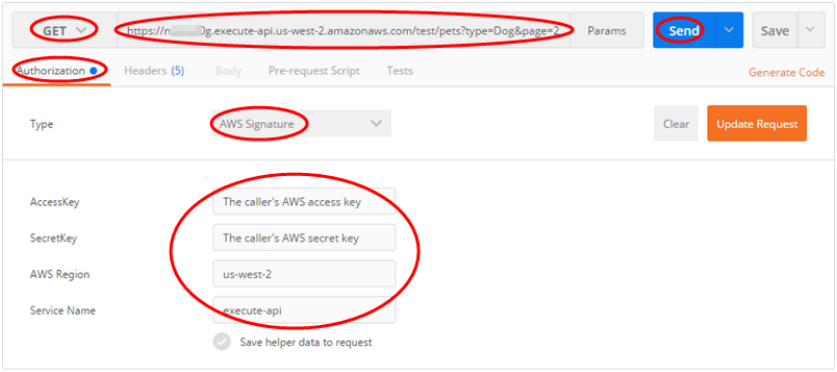
Once deployed, you can obtain the invocation URLs (**Invoke URL**) of the API's endpoints. For example, the GET on Pets method's invocation URL is as follows:



To invoke this API method in a client (e.g., a Postman browser), 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 **Authorization** type and then specify the required properties (as shown), following the Signature Version 4 protocols. Finally, send the request.



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.

Make Synchronous Calls to Lambda Functions

AWS Lambda provides an easy way to build back ends without managing servers. API Gateway and Lambda together can be powerful to create and deploy serverless Web applications. In this walkthrough, you learn how to create Lambda functions and build an API Gateway API to enable a Web client to call the Lambda functions synchronously. For more information about Lambda, see the AWS Lambda Developer Guide. For information about asynchronous invocation of Lambda functions, see Create an API as a Lambda Proxy (p. 143).

Topics

Step 1: Prerequisites

You must grant API Gateway access permission to the IAM user who will perform the tasks discussed here. The IAM user must have full access to work with Lambda. For this, you can use or customize the managed policy of **AWSLambdaFullAccess** (`arn:aws:iam::aws:policy/AWSLambdaFullAccess`) and attach it to the IAM user. For more information, see Get Ready to Use API Gateway (p. 4). The IAM user must also be allowed to create policies and roles in IAM. For this you can use or customize the managed policy of **IAMFullAccess** (`arn:aws:iam::aws:policy/IAMFullAccess` and attach it to the user.

Step 2: Create an API

In this step, you will create a new API named `MyDemoAPI`. To create the new API, follow the steps in Build an API Gateway API Step by Step (p. 14).

Step 3: Create a Resource

In this step, you will create a new resource named **`MyDemoResource`**. To create this resource, follow the steps in Build an API Gateway API Step by Step (p. 14).

Step 4: Create Lambda Functions

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

In this step, you will create two new Lambda functions. The first Lambda function, `GetHelloWorld`, will log the call to Amazon CloudWatch and return the JSON object `{"Hello": "World"}`. For more information about JSON, see Introducing JSON.

The second Lambda function, `GetHelloWithName`, will take an input ("name"), log the call to CloudWatch, and return the JSON object `{"Hello": user-supplied-input-value}`. If no input value is provided, the value will be `"No-Name"`.

You will use the Lambda console to create the Lambda functions and set up the required execution role/policy. You will then use the API Gateway console to create an API to integrate API methods with the Lambda functions; the API Gateway console will set up the required Lambda invocation role/policy. 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/policy for API Gateway to invoke the Lambda functions. For more information on how to set up Lambda invocation and execution roles, see Create Lambda Invocation and Execution Roles (p. 30). For more information about Lambda see AWS Lambda Developer Guide.

To create the GetHelloWorld Lambda function

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**.
   - If the **Lambda: Function list** page appears, choose **Create a Lambda function**.
3. From **Select blueprint**, select the **hello-world** blueprint for `nodejs`. You may need to type `Hello` as the search filter to bring the blueprint in focus.
4. For **Name**, type **`GetHelloWorld`**.
5. For **Description**, type **`Returns {"Hello":"World"}.`**
6. For **Runtime**, choose **Node.js** or leave as-is.
7. Under **Lambda function code**, replace the default code statements in the inline code editor with the following:

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

exports.handler = function(event, context) {
  console.log('"Hello":"World"');
  context.done(null, {"Hello":"World"});  // SUCCESS with message
};
```

Tip
The preceding code is written in Node.js. The `console.log` method writes information to an Amazon CloudWatch Log. The `event` parameter contains the event's data. The `context` parameter contains callback context. Lambda uses `context.done` to perform follow-up actions. 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*.

8. Under **Lambda function handler and role**, leave the default of `index.handler` for **Handler**.
9. For **Role**, choose `* Basic execution role` under **Create new role**.
   a. Leave the default selection of `lambda_basic_execution` for **IAM Role**.
   b. Leave the default selection of `Create a new Role Policy` for **Policy Name**.
   c. Choose **Allow**.
10. For **Advanced settings** leave the default setting as is.
11. Choose **Next**
12. Choose **Create function**.
13. For the newly created **GetHelloWorld** function, note the AWS region where you created this function. You will need it later.
14. To test the newly created function, as a good practice, choose **Actions** and then select **Configure test event**.
15. For **Input test event**, replace any default code statements with the following, and then choose **Save and test**.

```
{

}
```

Tip
This function does not use any input. Therefore, we provide an empty JSON object as the input.

16. Choose **Test** to invoke the function. The **Execution result** section shows `{"Hello": "World"}`. The output is also written to CloudWatch Logs.
17. Go to the **Functions** list to create the next Lambda function.

In addition to the Lambda function, an IAM role (`lambda_basic_execution`) is also created as the result of this procedure. You can view this in the IAM console. Attached to this IAM role is the following inline policy that grants users of your AWS account permission to call the CloudWatch `CreateLogGroup`, `CreateLogStreams`, and `PutLogEvents` actions on any of the CloudWatch resources.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

A trusted entity of this IAM role is `lambda.amazonaws.com`, which has the following 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 user to invoke the Lambda function and for Lambda to call the supported CloudWatch actions on the user's behalf.

To create the GetHelloWithName Lambda function

1. Choose **Create a Lambda function**.
2. From **Select blueprint**, select the **hello-world** blueprint for `nodejs`.
3. Type **`GetHelloWithName`** for **Name**.
4. For **Description**, type **`Returns {"Hello":", a user-provided string, and "}.`**
5. For **Runtime**, choose **Node.js**.
6. In the code editor under **Lambda function code** replace the default code statements with the following:

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

exports.handler = function(event, context) {
  var name = (event.name === undefined ? 'No-Name' : event.name);
  console.log('"Hello":"' + name + '"');
  context.done(null, {"Hello":name}); // SUCCESS with message
};
```

7. Under **Lambda function handler and role**, leave the default of `index.handler` for **Handler**.
8. For **Role**, choose **`lambda_basic_execution`** under **Use existing role**, assuming you have created the `lambda_basic_execution` role in the previous procedure.
9. Leave the default values for **Advanced settings**. Then choose **Next**.
10. Choose **Create function**.
11. For the newly created **GetHelloWorldName** function, note the AWS region where you created this function. You will need it in later steps.
12. To test this newly created function, choose **Actions** and then **Configure test event**.
13. In **Input test event**, replace any default code statements with the following, and then choose **Save and test**.

```
{
  "name": "User"
}
```

Tip
The function calls `context.name` to read the input name. We expect it to return `{"Hello": "User"}`, given the above input.

You can experiment with this function by removing `"name": "User"` from the **Input test event** for the function and choosing **Save and test** again. You should see the output of `{"Hello": "No-Name"}` under **Execution result** in the Lambda console, as well as in CloudWatch Logs.

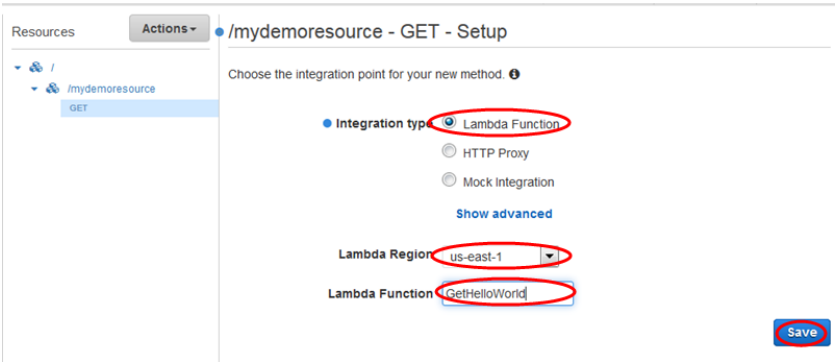
Step 5: Create and Test a GET Method

Switch back to the API Gateway console. In this step, you will create a GET method, connect it to your `GetHelloWorld` function in Lambda, and then test it. You use a GET method primarily to retrieve or read a representation of a resource. If successful, the GET method will return a JSON-formatted object.

To create and test the GET method

1. In the API Gateway console, from **APIs**, choose **MyDemoAPI**.
2. In the **Resources** pane, choose **/mydemoresource**. From **Actions**, choose **Create Method**. Choose `GET` from the HTTP method drop-down list and then choose the checkmark to create the method.
3. In the GET method **Setup** pane, for **Integration type**, choose **Lambda Function**. For **Lambda Region**, choose the region (.e.g, `us-east-1`) where you created the Lambda functions. In **Lambda Function**, type `GetHelloWorld`. Choose **Save** to finish setting up the integration request for this method.

   For a list of region names and identifiers, see AWS Lambda in the *Amazon Web Services General Reference*.



4. For **Add Permission to Lambda Function**, choose **OK**.
5. In the **Method Execution** pane, choose **TEST** from the **Client** box, and then choose the **Test** button. If successful, **Response Body** will display the following:

```
{
  "Hello": "World"
}
```

By default, API Gateway will pass through the request from the API caller. For the GET method call you just created, as well as for HEAD method calls, a Lambda function will receive an empty JSON response by default and then return the response from the Lambda function without modifications.

In the next step, you will create a POST method call. For POST and PUT method calls, you can pass in a request body in JSON format, which the Lambda function will receive as its input event. Optionally, you can transform the input to the Lambda function by using mapping templates in API Gateway.

Step 6: Create and Test a POST Method

In this step, you will create a new POST method, connect it to your `GetHelloWithName` function in Lambda, and then test it. If successful, the POST method typically returns to the caller the URI of the newly created resource. In this walkthrough, the POST method will simply return a JSON-formatted object.

To create and test the POST method

1. In the **Resources** pane, choose **/mydemoresource**, and then choose **Create Method**.
2. For the HTTP method, choose **POST**, and then choose the checkmark to save your choice.
3. In the **Setup** pane, for **Integration Type**, choose **Lambda Function**.
4. For **Lambda Region**, choose the region identifier that corresponds to the region name in which you created the `GetHelloWithName` Lambda function.
5. For **Lambda Function**, type **`GetHelloWithName`**, and then choose **Save**.
6. When you are prompted to give API Gateway permission to invoke your Lambda function, choose **OK**.
7. In the **Method Execution** pane, in the **Client** box, and then choose **TEST**. Expand **Request Body**, and type the following:

```
{
  "name": "User"
}
```

8. Choose **Test**. If successful, **Response Body** will display the following:

```
{
  "Hello": "User"
}
```

9. Change **Request Body** by removing **`"name": "User"`** so that only a set of curly braces (`{ }`) remain, and then choose **Test** again. If successful, **Response Body** will display the following:

```
{
  "Hello": "No-Name"
}
```

The API Gateway console-assisted Lambda function integration uses the AWS service proxy integration type for Lambda. It streamlines the process to integrate an API method with a Lambda function by setting up, among other things, the required Lambda function invocation URI and the invocation role on behalf of the API developer.

The `GET` and `POST` methods discussed here are both integrated with a `POST` request in the back end:

```
POST /2015-03-31/functions/FunctionArn/invocations?Qualifier=Qualifier HTTP/1.1
```

```
X-Amz-Invocation-Type: RequestReponse
...
Content-Type: application/json
Content-Length: PayloadSize

Payload
```

The `X-Amz-Invocation-Type: RequestReponse` header specifies that the Lambda function be invoked synchronously. *`FunctionArn`* is of the `arn:aws:lambda:`*`region`*`:`*`account-id`*`:function:`*`FunctionName`* format. In this walkthrough, the console sets *`FunctionName`* as **`GetHelloWorld`** for the `GET` method request and supplies an empty JSON payload when you test-invoke the method. For the `POST` method, the console sets *`FunctionName`* as **`GetHelloWithName`** and passes the caller-supplied method request payload to the integration request. You can regain full control of a method creation and setup by going through the AWS service proxy integration directly. For more information, see Create an API as a Lambda Proxy (p. 143).

Step 7: Deploy the API

You are now ready to deploy your API so that you can call it outside of the API Gateway console. In this step, you will create a stage. In API Gateway, a stage defines the path through which an API deployment is accessible. For example, you can define a `test` stage and deploy your API to it, so that a resource named `MyDemoAPI` is accessible through a URI that ends in `.../test/MyDemoAPI`.

To deploy the API

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 **`test`**.

   Note
   The input must be UTF-8 encoded (i.e., unlocalized) text.
4. For **Stage description**, type **`This is a test.`**
5. For **Deployment description**, type **`Calling Lambda functions walkthrough.`**
6. Choose **Deploy**.

Step 8: Test the API

In this step, you will go outside of the API Gateway console to call the GET and POST methods in the API you just deployed.

To test the GET-on-mydemoresource method

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

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/test
   ```
2. In a separate web browser tab or window, paste the URL into the address box. Append the path to your resource (**`/mydemoresource`**) to the end of the URL. The URL should look something like this:

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

3. Browse to this URL. If the GET method is successfully called, the web page will display:

```
{"Hello":"World"}
```

To test the POST-on-mydemoresource method

1. You will not be able to test a POST method request with your web browser's address bar. Instead, use an advanced REST API client, such as Postman, or the cURL command-line tool.
2. Send a POST method request to the URL from the previous procedure. The URL should look something like this:

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

Be sure to append to the request headers the following header:

```
Content-Type: application/json
```

Also, be sure to add the following code to the request body:

```
{
  "name": "User"
}
```

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 "{\"name\": \"User\"}"
 https://my-api-id.execute-api.region-id.amazonaws.com/test/mydemoresource
```

If the POST method is successfully called, the response should contain:

```
{"Hello":"User"}
```

Step 9: 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.

Caution
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 Identity and Access Management (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.

Next Steps

You may want to proceed to the next walkthrough, which shows how to map header parameters from the method request to the integration request and from the integration response to the method response. It uses the HTTP proxy integration to connect your API to HTTP endpoints in the back end.

For more information about API Gateway, see What Is Amazon API Gateway? (p. 1). For more information about REST, see RESTful Web Services: A Tutorial.

Create Lambda Invocation and Execution Roles

Before you create AWS Lambda functions, you must assign appropriate permissions for the functions to execute the specified Amazon CloudWatch action (namely, writing to the CloudWatch Log) and for API Gateway to invoke the Lambda functions. You set up the permissions using IAM roles and policies for API Gateway to invoke your code and for Lambda to execute your code. For more information about invocation and execution roles/policies in Lambda see Permission Model in the *AWS Lambda Developer Guide*.

To create the Lambda invocation role and its policy

1. Open the IAM console at https://console.aws.amazon.com/iam/.

   If you are using the IAM-managed **AWSLambdaRole** policy, skip to Step 8 to create an invocation role.
2. In **Details**, choose **Policies**.
3. Do one of the following:

   - If a list of policies appears, choose **Create Policy**.
   - If the **Welcome to Managed Policies** page appears, choose **Get Started**, and then choose **Create Policy**.

4. For **Create Your Own Policy**, choose **Select**.

5. For **Policy Name**, type a name for the policy; for example, **`APIGatewayLambdaInvokePolicy`**.
6. For **Description**, type **`Enables API Gateway to call Lambda functions.`**
7. For **Policy Document**, type the following, and then choose **Create Policy**.

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

8. In **Details**, choose **Roles**.
9. Choose **Create New Role**.
10. For **Role Name**, type a name for the invocation role; for example, **`APIGatewayLambdaInvokeRole`**, and then choose **Next Step**.
11. Under **Select Role Type**, with the option button next to **AWS Service Roles** already chosen, for **Amazon API Gateway**, choose **Select**.
12. For **Attach Policy**, if the policy you want is in the list, choose it before choosing **Next Step**. Otherwise, simply choose **Next Step** to proceed.
13. For **Role ARN**, make a note of the invocation role's Amazon Resource Name (ARN). You will need this ARN in later steps when you specify the invocation role explicitly. The ARN should look similar to this: `arn:aws:iam::123456789012:role/APIGatewayLambdaInvokeRole`, where `123456789012` is your AWS account ID.
14. Choose **Create Role**.

    The newly created IAM role will have the following trust policy.

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

    The preceding policy document enables API Gateway to assume roles taken up by and, hence, take actions on behalf of your AWS account.

To create the Lambda execution role and its policies

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In **Details**, choose **Policies**.
3. Choose **Create Policy**.
4. For **Create Your Own Policy**, choose **Select**.
5. For **Policy Name**, type a name for the policy (for example, **`APIGatewayLambdaExecPolicy`**).
6. For **Description**, type **`Enables Lambda to execute code.`**
7. For **Policy Document**, type the following, and then choose **Create Policy**.

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

Note
The preceding policy document permits all log actions on Amazon CloudWatch Logs. Typically, you would add other permissions required by your Lambda function to interact with AWS services, such as uploading an object to an Amazon S3 bucket. In this walkthrough, the Lambda functions you create are very simple; they do not interact with AWS services.

8. In **Details**, choose **Roles**.
9. Choose **Create New Role**.
10. In **Role Name**, type a name for the execution role (for example, **`APIGatewayLambdaExecRole`**), and then choose **Next Step**.
11. Next to **AWS Lambda**, choose **Select**.

Note
IAM will attach the following resource-policy document in **Trust Relationships**:

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

This policy document enables Lambda to assume roles taken up by and, hence, to take actions on behalf of your AWS account.

Map Request Parameters for an API Gateway API as an HTTP Proxy

In this walkthrough, you will learn how to map method request parameters to the corresponding integration request parameters for an API Gateway API. As an illustration, we will create an example API as an HTTP proxy and use it to demonstrate how to use API Gateway to map a method request parameter to the corresponding integration request parameter and to access the publicly accessible HTTP endpoint of:

```
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 hit the `Enter` or `Return` key, you will 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 above endpoint can take two query parameters: `type` and `page`. For example, if you change the above URL to the following:

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

you will 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, if you 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 also take POST requests with a payload. For example, if you use Postman to send a POST method request to the following:

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

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

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

you will receive 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 as an HTTP proxy of this PetStore website. The tasks includes 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 back end website, pass the payload without modifications through the integration request to the back-end HTTP endpoint.

Topics

Prerequisites

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

1. Complete the steps in Get Ready to Use API Gateway (p. 4), including assigning API Gateway access permission to the IAM user.
2. At a minimum, follow the steps in Build an API Gateway API Step by Step (p. 14) to create a new API named `MyDemoAPI` in the API Gateway console.

Step 1: Create Resources

In this step, you will create three resources that will 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`**.

   This maps to `petstore` in the HTTP endpoint.
3. For **Resource Path**, accept the default of **/petstorewalkthrough**, and then choose **Create Resource**.

   This maps to `/petstore` in the HTTP endpoint.

To create the second resource

1. In the **Resources** pane, choose **/petstorewalkthrough**, and then choose **Create Resource**.
2. For **Resource Name**, type **`pets`**.

   This maps to `pets` in the HTTP endpoint.
3. For **Resource Path**, accept the default of **/petstorewalkthrough/pets**, and then choose **Create Resource**.

   This maps to `/petstore/pets` in the HTTP endpoint.

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}`**. Be sure to 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 GET and POST Methods

In this step, you will create two GET methods and a POST method to interact with the HTTP endpoint.

To create the first GET method

1. In the **Resources** pane, choose **/petstorewalkthrough/pets**, and then choose **Create Method** from the **Actions** drop-down menu.
2. For the HTTP method, choose **GET**, and then save your choice.

To create the second GET method

1. In the **Resources** pane, choose **/petstorewalkthrough/pets/{petId}**, and then choose **Create Method**.
2. For the HTTP method, choose **GET**, and then save your choice.

To create the POST method

1. In the **Resources** pane, choose **/petstorewalkthrough/pets** again, and then choose **Create Method**.
2. For the HTTP method, choose **POST**, and then save your choice.

Step 3: Set Up and Test the Methods

In this step, you will integrate the methods with the back-end 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:

- Integrate the method request of `GET /petstorewalkthrough/pets` with the integration request of `GET https://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, in **/petstorewalkthrough/pets**, choose **GET**.
2. In the **Setup** pane, for **HTTP method**, choose **GET**.
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 **Create a new query string** (the check mark icon).
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 **Create a new query string**.
12. Choose **Method Execution**, choose **Integration Request**, and then choose the arrow next to **URL Query String Parameters**.
13. Choose **Add query string**.
14. For **Name**, type **`type`**.
15. 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.
16. Choose **Create** (the check mark icon).
17. Choose **Add query string** again.
18. For **Name**, type **`page`**.
19. 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.
20. Choose **Create**.
21. Choose **Method Execution**, and in the **Client** box, choose **TEST**. In the **Query Strings** area, for **petType**, type **`cat`**. For **petsPage**, type **2**.
22. Choose **Test**. If successful, **Response Body** will display 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:

- Integrate the method request of `GET /petstorewalkthrough/pets/{petId}` with the integration request of `GET https://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, in **/petstorewalkthrough/pets/{petId}**, choose **GET**.
2. In the **Setup** pane, for **HTTP method**, choose **GET**.
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 **Create**.
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** will display the following:

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

To set up and test the POST method

This procedure demonstrates the following:

- Integrate the method request of `POST /petstorewalkthrough/pets` with the integration request of `POST https://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, in **/petstorewalkthrough/pets**, choose **POST**.
2. In the **Setup** pane, for **HTTP method**, choose **POST**.
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** will display the following:

```
{
  "pet": {
```

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

Step 4: Deploy the API

In this step, you will 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 5: Test the API

In this step, you will 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/petstorewalk
   through/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/petstorewalk
through/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/petstorewalk
through/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. Be sure to append `/petstorewalkthrough/pets` so that it looks like this:

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

   Also, be sure to append the following header:

```
Content-Type: application/json
```

And be sure to 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.amazon
aws.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

You may want to begin the next walkthrough, which shows you how to use models and mappings in API Gateway to transform the output of an API call from one data format to another. See Transform Response Payload (p. 41).

Use Models and Mapping Templates to Transform Response Payload

In this walkthrough, you will learn 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 Call Lambda Functions Synchronously (p. 22) and the Map Request Parameters (p. 33). If you have not yet completed those walkthroughs, we suggest you do them first.

In this walkthrough, you will use API Gateway to get example data from a publicly-accessible HTTP endpoint and from an AWS Lambda function you will 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 Set Up Request and Response Payload Mappings (p. 72).

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

Prerequisites

Before you begin this walkthrough, you should have already done the following:

1. Complete the steps in Get Ready to Use API Gateway (p. 4), including assigning API Gateway access permission to an IAM user.
2. Open the API Gateway console and create a new API named `MyDemoAPI`. For more information, see Build an API Gateway API Step by Step (p. 14).
3. Create two resources: `petstorewalkthrough` and `pets`. For more information, see Create Resources (p. 35) in the Map Request Parameters (p. 33).
4. To use the Lambda portions of this walkthrough, make sure the IAM user has full access to work with Lambda. You can use the IAM console to attach the **AWSLambdaFullAccess** AWS managed policy to the IAM user.
5. Make sure the IAM user has access to create policies and roles in IAM. If you have not done so already, create a Lambda execution role named `APIGatewayLambdaExecRole` in IAM. For more information, see Create Lambda Functions (p. 23) in the Call Lambda Functions Synchronously (p. 22).

Step 1: Create Models

In this step, you will 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 will create four resources. The first three resources will enable you to get the example data from the HTTP endpoint in the three output formats. The last resource will enable 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 will 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. For the HTTP method, choose **GET**, and then save your choice.

To create the second GET method

1. In the **Resources** list, choose **/petstorewalkthrough/lambdaflattensome**, and then choose **Create Method**.
2. For the HTTP method, choose **GET**, and then save your choice.

To create the third GET method

1. In the **Resources** list, choose **/petstorewalkthrough/flattensome**, and then choose **Create Method**.

2. For the HTTP method, choose **GET**, and then save your choice.

To create the fourth GET method

1. In the **Resources** list, choose **/petstorewalkthrough/noflatten**, and then choose **Actions**, **Create Method**.
2. For the HTTP method, choose **GET**, and then save your choice.

Step 4: Create a Lambda Function

In this step, you will 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) {
  context.done(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 event's data. `context` contains callback context. Lambda uses `context.done` to perform follow-up actions. 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 Call Lambda Functions Synchronously (p. 22).
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 will 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 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.
15. Choose **Go to function list**.

Step 5: Set Up and Test the Methods

In this step, you will specify the URL and data output schema for the three GET methods associated with the HTTP endpoint, testing each method as you proceed. You will also specify the Lambda function name, data input schema, and data output schema for the GET method associated with the Lambda function. You will then test this method.

To specify settings for the first GET method and then test it

1. In the **Resources** pane, in **/petstorewalkthrough/flattenall**, choose **GET**.
2. For **HTTP Method**, choose **GET**.
3. In the **Setup** pane, for **Endpoint URL**, type **`http://petstore-demo-endpoint.execute-api.com/petstore/pets`**.
4. Choose **Save**.
5. In the **Method Execution** pane, choose **Method Response**, and then choose the arrow next to **200**.
6. In the **Response Models for 200** area, for **application/json**, choose **Edit**. For **Models**, choose **PetsModelFlattenAll**, and then choose **Save**.
7. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200**.
8. In the **Template Mappings** area, for **Content type**, choose **application/json**, and then choose **Edit**. Clear **Output passthrough**. For **Generate template from model**, choose **PetsModelFlattenAll**. This displays the **PetsModelFlattenAll** model as a starting point.
9. Modify the 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
  ]
}
```

10. Choose **Update**.
11. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** will display 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 specify settings for the second GET method and then test it

1. In the **Resources** pane, in **/petstorewalkthrough/lambdaflattensome**, choose **GET**.
2. In the **Setup** pane, for **Execution Type**, choose **Lambda Function**.
3. For **Lambda Region**, choose the region identifier that corresponds to the region in which you created the `GetPetsInfo` Lambda function. For example, if you created this Lambda in the US East (N. Virginia) region, you would choose **`us-east-1`**. For a list of region names and identifiers, see AWS Lambda in the *Amazon Web Services General Reference*.
4. For **Lambda Function**, type **`GetPetsInfo`**, and then choose **Save**.
5. When you are prompted to give API Gateway permission to invoke your Lambda function, choose **Ok**.
6. In the **Method Execution** pane, choose **Integration Request**.
7. Next to **Templates**, choose **Add**.
8. For **Content-Type**, type **application/json**.
9. Leave **Input passthrough** cleared. For **Generate template from model**, choose **PetsLambdaModel**. This displays the **PetsLambdaModel** model as a starting point.
10. In the **Input mapping** area, modify the code as follows, and then choose **Update**:

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

#end
]
```

11. Choose **Method Execution**, choose **Method Response**, and then choose the arrow next to **200**.
12. In the **Response Models for 200** area, for **application/json**, choose **Edit**. For **Models**, choose **PetsModelFlattenSome**, and then choose **Save**.
13. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200**.
14. In the **Template Mappings** area, for **Content type**, choose **application/json**, and then choose **Edit**. Clear **Output passthrough**. For **Generate template from model**, choose **PetsModelFlattenSome**. This displays the **PetsModelFlattenSome** model as a starting point.
15. Modify the code as follows, and then choose **Update**:

```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
```

```
  {
    "description" : "Item $elem.id is a $elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end

#end
]
```

16. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** will display 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 specify settings for the third GET method and then test it

1. In the **Resources** pane, in **/petstorewalkthrough/flattensome**, choose **GET**.
2. In the **Setup** pane, for **HTTP method**, choose **GET**.
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 Response**, and then choose the arrow next to **200**.
6. In the **Response Models for 200** area, for **application/json**, choose **Edit**. For **Models**, choose **PetsModelFlattenSome**, and then choose **Save**.
7. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200**.
8. In the **Template Mappings** area, for **Content type**, choose **application/json**, and then choose **Edit**. Clear **Output passthrough**. For **Generate template from model**, choose **PetsModelFlattenSome**. This displays the **PetsModelFlattenSome** model as a starting point.
9. 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
]
```

10. Choose **Update**.
11. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** will display 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 specify settings for the fourth GET method and then test it

1. Return to the API Gateway console.
2. If **MyDemoAPI** is displayed, choose **Resources**.
3. In the **Resources** pane, in **/petstorewalkthrough/noflatten**, choose **GET**.
4. In the **Setup** pane, for **HTTP method**, choose **GET**.
5. For **Endpoint URL**, type **`http://petstore-demo-endpoint.execute-api.com/petstore/pets`**.
6. Choose **Save**.
7. In the **Method Execution** pane, choose **Method Response**, and then expand **200**.
8. In the **Response Models for 200** area, for **application/json**, choose **Edit**. For **Models**, choose **PetsModelNoFlatten**, and then choose **Save**.
9. Choose **Method Execution**, choose **Integration Response**, and then choose the arrow next to **200**.
10. In the **Template Mappings** area, for **Content type**, choose **application/json**, and then choose **Edit**. Clear **Output passthrough**. For **Generate template from model**, choose **PetsModelNoFlatten**. This displays the **PetsModelNoFlatten** model as a starting point.
11. 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
]
```

12. Choose **Update**.
13. Choose **Method Execution**, and in the **Client** box, choose **TEST**, and then choose **Test**. If successful, **Response Body** will display 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 will 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 will 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/petstorewalk
   through/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.

> **Caution**
> 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 Identity and Access Management (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 Create an AWS Service Proxy (p. 55).

Create an AWS Service Proxy for Amazon SNS

In this walkthrough, you will learn how to use API Gateway to connect a custom API to an AWS service through what we call an AWS service proxy. This enables you to call an AWS service directly instead of through an AWS Lambda function. An AWS service proxy can call only one action in an AWS service, and that action typically does not change. If you want more flexibility, you should call a Lambda function instead.

This walkthrough builds on the instructions and concepts in the Call Lambda Functions Synchronously (p. 22), which shows you how to use API Gateway to create a custom API, connect it to a set of AWS Lambda functions, and then call the Lambda functions from your API. If you have not yet completed that walkthrough, we suggest that you do it first.

Topics

Prerequisites

Before you begin this walkthrough, you should have already done the following:

1. Complete the steps in Get Ready to Use API Gateway (p. 4).
2. Make sure the IAM user has access to create policies and roles in IAM. You will need to create an IAM policy and role in this walkthrough.
3. At a minimum, open the API Gateway console and create a new API named `MyDemoAPI`. For more information, see Build an API Gateway API Step by Step (p. 14).
4. Deploy the API at least once to a stage named `test`. For more information, see Deploy the API (p. 28) in the Call Lambda Functions Synchronously (p. 22).
5. Complete the rest of the steps in the Call Lambda Functions Synchronously (p. 22).
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 will create a resource that will enable 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 will create a GET method that will enable 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 will create an IAM role that your AWS service proxy will use 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 will 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 Identity and Access Management (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"
      ]
    }
```

```
  ]
}
```

Note
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 New Role**.
10. For **Role Name**, type a name for the execution role (for example, **`APIGatewayAWSProxyExecRole`**), and then choose **Next Step**.
11. Next to **Amazon EC2**, choose **Select**.

    Note
    You choose **Select** here because you need to choose a standard AWS service role statement before you can continue. There is currently no option to choose a standard API Gateway service role statement. Later in this step, you will modify the standard Amazon EC2 service role statement for use with API Gateway.

12. In the list of policies, select **APIGatewayAWSProxyExecPolicy**, and then choose **Next Step**.
13. For **Role ARN**, make a note of the Amazon Resource Name (ARN) for the execution role. You will need it later. The ARN should look similar to: `arn:aws:iam::123456789012:role/APIGatewayAWSProxyExecRole`, where `123456789012` is your AWS account ID.
14. Choose **Create Role**.

    The invocation role IAM just created enables Amazon EC2 to get a list of the Amazon SNS topics for the AWS account. You will change this role to enable API Gateway to get a list of the Amazon SNS topics for the AWS account instead.
15. In the list of roles, select **APIGatewayAWSProxyExecRole**.
16. In the **Trust Relationships** area, choose **Edit Trust Relationship**.
17. For **Policy Document**, replace `ec2.amazonaws.com` with `apigateway.amazonaws.com` so that the access control policy document now looks as follows:

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

    This policy document enables API Gateway to take actions on behalf of your AWS account.
18. Choose **Update Trust Policy**.

Step 4: Specify Method Settings and Test the Method

In this step, you will specify the settings for the GET method so that it can interact with an AWS service through an AWS service proxy. You will 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** will display 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 will deploy the API so that you can begin calling 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 will 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":[{"Topi
   cArn": "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":{"Re
   questId":"abc1de23-45fa-6789-b0c1-d2e345fa6b78"}}}
   ```

Step 7: Clean Up

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

> **Caution**
> 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 Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. In the **Details** area, click **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. 125), Create an API as a Lambda Proxy (p. 143) or Create an API as an Amazon Kinesis Proxy (p. 158).

Creating an API in Amazon API Gateway

Topics

Create an API in API Gateway

In Amazon API Gateway you can create an API using the API Gateway console, AWS CLI, the API Gateway control service REST API, and platform-specific or language-specific SDKs.

Topics

Create an API Using the API Gateway Console

To create an API Gateway API using the API Gateway console, see Build an API Gateway API Step by Step (p. 14).

You can learn how to create an API by following an example. For more information, see Build and Test an API Gateway API from an Example (p. 6).

Alternatively, you can create an API by using the API Gateway Import API (p. 110) feature to upload an external API definition, such as one expressed in the Swagger 2.0 with the API Gateway Extensions to Swagger (p. 115). The example provided in Build and Test an API Gateway API from an Example (p. 6) uses the Import API feature.

Create an API Using the API Gateway Control Service API

For more information about the API Gateway Control Service API, see Amazon API Gateway REST API Reference.

Create an API Using the AWS SDK for API Gateway

For more information using a AWS SDK, see AWS SDKs.

Create an API Using the AWS CLI

For an example of creating an API Gateway API Using AWS CLI, see Create an API Gateway API for Lambda tutorial.

Set up API Gateway API Method and Integration

Before Configuring Methods

- You must have the method available in API Gateway. Follow the instructions in Build an API Gateway API Step by Step (p. 14).
- 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 and created the Lambda function with which your method will communicate in AWS Lambda. To create the roles and function, use the instructions in Step 4: Create Lambda Functions (p. 23) of the Call Lambda Functions Synchronously (p. 22).
- If you want the method to communicate with an HTTP proxy, you must have already created and have access to the HTTP endpoint URL with which your method will communicate.

After Setting Up Methods and Integration

The next step is to deploy the API to make it open for access. For instructions, see Deploying an API (p. 221).

Topics

Configure How API Gateway Integrates the Method with a Back End

The settings of an API method defines the method and describes its behaviors. To create a method, you must specify a resource, including the root ("/"), on which the method is exposed, a method type (GET, POST, etc.), and how it will be integrated with the targeted back end. 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 integration request and response specifies how API Gateway interacts with their back end: enforcing secure communications over HTTPS with the back end and translating data formats between the client and back end. The following topics describe how to use the API Gateway console to specify a 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 communicating with a Lambda function.
   - Choose **HTTP Proxy** if your API will be communicating with an HTTP endpoint.
   - Choose **Show Advanced**, **AWS Service Proxy** if your API will be communicating directly with an AWS service.
   - Choose **Mock Integration** if your API is not yet final, but you want to generate API responses from API Gateway anyway to unblock dependent teams for testing. If you choose this option, skip the rest of the instructions in this topic and see Configure Mock Integration for a Method (p. 69).

3. If you chose **Lambda Function**, do the following:

   1. 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, you would choose **`us-east-1`**. For a list of region names and identifiers, see AWS Lambda in the *Amazon Web Services General Reference*.
   2. For **Lambda Function**, type the name of the Lambda function, and then choose the function's corresponding ARN.
   3. Choose **Save**.

4. If you chose **HTTP Proxy**, do the following:

   1. For **HTTP method**, choose the HTTP method type that most closely matches the method in the HTTP proxy.
   2. For **Endpoint URL**, type the URL of the HTTP proxy you want this method to use.
   3. Choose **Save**.

5. If you chose **Mock Integration**, do the following:

   - Choose **Save**.

6. If you chose **Show advanced**, **AWS Service Proxy**, do the following:

   1. For **AWS Region**, choose the AWS region you want this method to use to call the action.
   2. For **AWS Service**, choose the AWS service you want this method to call.

3. 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**.
4. 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**.
5. For **Execution Role**, type the ARN of the IAM role 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. 23) section of the Call Lambda Functions Synchronously (p. 22); and 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 actions on behalf of your AWS account:

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

6. If the action you typed for **Action** provides a custom resource path 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**.
7. Choose **Save**.

7. Do both of the following:

- Specify how the method will receive requests from, and send responses to, callers (which API Gateway refers to as the API's method request/response), and how the method will authorize requests by following the instructions in Configure How a User Calls an API Method (p. 65).
- Specify how the method will send requests to, and receive responses from, the Lambda function, HTTP proxy, or AWS service proxy (which API Gateway refers to as the API's integration request/response) by following the instructions in Configure How Data Is Mapped between Method and Integration (p. 67).

Configure How an API User Calls an API Method in Amazon API Gateway

To use the API Gateway console to specify an API's method request/response and the way in which the method will authorize requests, follow these instructions.

Note
These instructions assume you have already completed the steps in Configure How a Method Is Integrated with a Back End (p. 63).

1. With the method selected in the **Resources** pane, in the **Method Execution** pane, choose **Method Request**.
2. To assign custom access permissions to the method, in the **Authorization Settings** area, for **Authorization Type**, choose **Edit**, and then choose **AWS_IAM**. Only IAM roles with the correct IAM policy attached will be allowed to call this method. If you do not want to assign custom access permissions to the method, choose **NONE**.

   - 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.

   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. 23) section of the Call Lambda Functions Synchronously (p. 22).

   To save your choice, choose **Update**. Otherwise, choose **Cancel**.

Note
You can also enable an API key. For instructions, see Use an API Key with the API Gateway Console (p. 203).

3. 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 **Create a new query string**.

   Note
   To remove the query string parameter, choose **Cancel** or **Remove**.
   To change the name of the query string parameter, you must remove it and create a new one.

4. 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.
   c. Optionally, check the **Caching** option to make this method as an API cache key. For more information, see Use Method/Integration Parameters as Cache Keys (p. 230).
   d. Choose **Create**

   Tip
   To remove the header parameter, choose **Cancel** or **Remove**.
   To change the name of the header parameter, you must remove the old header parameter and create a new one in its place.

5. For non-GET method types, expand **Request Models**, and for **Content Type** and **Model name**, type the content type and choose the name of a model that will transform caller-supplied data into the expected format.

   To create a model, see Create a Model (p. 79).

6. To send a set of custom response headers, a custom response data format, or both, back to callers based on the HTTP status code returned by the method, do the following:

   a. In the **Method Execution** pane, choose **Method Response**. By default, `200` response is included in the method responses. You can modify it, e.g., to have the method return `201` instead. In addition, you can add other responses, e.g., `409` for access denial and `500` for uninitialized stage variables used. Either choose the arrow icon next to **200** to specify settings for the 200 response, or choose **Add Response** to specify settings for any other HTTP response status code. If you choose **Add Response**, for **HTTP Status**, choose the response, choose **Create**, and choose the arrow next to the response.

      Tip
      You will use **Method Response** to specify all possible response codes for your API and use **Integration Response** to indicate to API Gateway how back-end errors are mapped to an HTTP status code.

   b. For each custom header you want to include in the response, in the **Response Headers** area, choose **Add Header**, type the name of the header, and then choose **Save**. (Choose **Remove** to remove a header from this list.)

      To specify a response model to transform the output's data from one format to another, in the **Response Models** area, choose **Add Response Model**. Type the content type (for **Content type**), choose the model's name (for **Models**), and then choose **Save**. Choose **Add Response Model** to specify an additional model, or choose **Create a model** to define a new model. (Choose **Remove** to remove a response model selection from this list.)

Configure How Data Is Mapped between a Method and its Integration in Amazon API Gateway

Note
API Gateway does not currently support binary payloads. Binary data can be passed around in a payload as a JSON property value of a Base64-encoded string.

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 Configure How a Method Is Integrated with a Back End (p. 63).

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.

      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 back end 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 back end 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. 108).

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 Set Up Request and Response Payload Mappings (p. 72).
5. You can map an integration response from the back-end 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 back-end 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 `context.fail(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**.

Configure Mock Integration for a Method 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 back end. As an API developer, you can use this feature to unblock other dependent teams needing to work with an API before the project development is complete. You can also leverage this feature to provision a landing page of 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 and Test an API Gateway API from an Example (p. 6).

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. The tasks involve setting up a mapping template in the integration request to specify a supported status code in the request payload and setting up maping templates, one for a supported status code, in the integration response to provide associated response payloads. At run time API Gateway retrieves the status code from the request payload and invokes the matching template to return the associated response payload. The integration request payload's content type must be `application/json` and its format must be of `{"statusCode": `*`ddd`*`, ... }`, where *`ddd`* stands for an HTTP status code. The integration response payload's content type can be any of those matching the response data, including `application/json`, `application/xml`, `text/html`, `text/plain` and etc.

In this section, you will learn how to use the API Gateway console to enable the mock integration for an API method.

Topics

Prerequisites

- You must have the method available in API Gateway. Follow the instructions in Build an API Gateway API Step by Step (p. 14).

Enable Mock Integration on a Method

1. Choose an API resource and create a method. In the **Setup** pane, choose **Mock Integration** , and then choose **Save**.
2. In the **Method Execution** pane, choose **Integration Request**.
3. By default, mock integrations return a 200 HTTP status code. To customize this default behavior, do the following:

    1. Expand **Mapping Templates**.
    2. For **Content-Type**, do one of the following:

        - If the desired content type is already visible (for example, **application/json**), then choose it.
        - If the desired content type is not already visible, then choose **Add mapping** template, type the desired content type (for example, **`application/json`**), and then choose **Create**.

    3. In the **Template** editor, type the content of the template you want API Gateway to use to determine which HTTP status code to use in the integration response. The template must output a JSON payload containing the `statusCode` property. For more information, see Example Request Templates (p. 71).
    4. Next to **Mapping template**, choose **Save**.

4. For each query string parameter or header parameter you want to add to the method, do the following:

    1. Choose **Method Execution**, and then choose **Method Request**.
    2. Choose the arrow next to **URL Query String Parameters** or **HTTP Request Headers**, and then choose **Add query string** or **Add header**, respectively.
    3. For **Name**, type the name of the query string parameter or header parameter, and then choose **Create a new query string** or **Create**, respectively.

        Note
        To remove a query string parameter or header parameter, choose **Cancel** or **Remove**. To change the name of a query string parameter or header parameter, you must remove it and create a new one in its place.

5. Choose **Method Execution**, and then choose **Method Response**.
6. Do one of the following:

    - If all of the **HTTP Status** entries you want to use are already visible (for example, **200**), then skip ahead to step 8.
    - If any of the **HTTP Status** entries you want to use are not already visible, then for each missing **HTTP Status** entry, choose **Add Response**, choose the HTTP status code that you want to use, and then choose **Create**.

7. Choose **Method Execution**, and then choose **Integration Response**.
8. Do one of the following:

- If all of the **Method response status** entries you want to use are already visible (for example, **200**), then skip ahead to step 10.
- If any of the **Method response status** entries you want to use are not already visible, then for each missing **Method response status** entry, choose **Add integration response**, for **Method response status** choose the **HTTP Status** entry you created earlier, and then choose **Save**.

9. For each **Method response status** entry you want to use, do the following:

   1. Expand the row that corresponds to the **Method response status** entry you want to use.
   2. For **HTTP status regex**, type the matching **HTTP Status** entry (for example, type **`400`** for a 400 **HTTP Status** entry or **`500`** for a 500 **HTTP Status** entry). Or specify a range of matching HTTP status codes (for example, **`5/d{2}`** matches all 5XX HTTP status codes).
   3. Expand **Mapping Templates**.
   4. For **Content-Type**, do one of the following:

      - If the desired content type is already visible (for example, **application/json**), then choose it.
      - If the desired content type is not already visible, then choose **Add mapping template**, type the desired content type (for example, **`application/json`**), and then choose **Create**.

   5. In the **Template** editor, type the contents of the template that you want API Gateway to use to respond to the caller. For more information, see Example Response Templates (p. 72).
   6. Next to **Mapping template**, choose **Save**.

10. Do one of the following to test the method:

    - Call the method from the API Gateway console. Follow the instructions in Test a Method Using the Console (p. 259).
    - Call the method from a web browser, a web debugging proxy tool or the cURL command-line tool, or from your own API. Follow the instructions in Calling a Deployed API (p. 258).

Example Request Templates

The following example shows a request template that always uses the 200 HTTP status code.

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

The following example shows a request template that uses the 200 HTTP status code if the request specifies the `petType` parameter of `cat`; 400 if the request specifies `dog`; and uses 500 otherwise. This example is based on the one in the Map Request Parameters (p. 33).

```
{
  #if( $input.params('petType') == "cat" )
    "statusCode": 200
  #elseif( $input.params('petType') == "dog" )
    "statusCode": 400
  #else
    "statusCode": 500
```

```
  #end
}
```

Example Response Templates

The following two examples show response templates that respond with the same information every time. These examples are based on the one in the Map Request Parameters (p. 33).

```
## Example 400 response.
{
  "Message": "Error: petType not valid."
}
```

```
## Example 500 response.
{
  "Message": "Error: petType not valid or not specified."
}
```

The following example shows a response template that responds with the same information every time, but includes the value the caller specified for the `petType` parameter. This example is based on the one in the Map Request Parameters (p. 33).

```
## Example 200 response for ?petType=cat (response will contain "type": "cat").
{
   "id": 1,
   "name": "Kitty",
   "type": "$input.params('petType')"
}
```

Set Up Amazon API Gateway API Request and Response Payload 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 back end. Similarly, the back end may return an integration response payload different from the method response payload, as expected by the front end. API Gateway lets you map the payload from a method request to the corresponding integration request and from an integration response to the corresponding method response. You use mapping templates to specify the mapping and can create model to facilitate the template generation. The section explains how to use the map the API request and response payload using models and mapping templates.

Topics

Models

In API Gateway, a model defines the format, also known as the schema or shape, of some data. Models are most useful for generating strongly typed SDK of your API. They can also be useful in helping generate a mapping template or validate a payload. Because API Gateway is designed to work primarily with JavaScript Object Notation (JSON)-formatted data, API Gateway uses JSON Schema to define the expected schema of the data.

For example, the following expresses some JSON data:

```
{
  "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
    }
  ]
}
```

In the preceding example:

- The top-level or root object contains a `department` string object, a `categories` array, and a `bins` array.
- The `categories` array contains a collection of string values.
- The `bins` array contains a collection of objects. Each object contains a `category` string object, a `type` string object, a `price` number object, a `unit` string object, and a `quantity` number object.

The corresponding model is expressed in JSON Schema notation:

```
{
  "$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:

- 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 `fugvjdxtri`:

```
{
  "$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/fugvjdxtri/mod
els/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.

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

- Input Model (Photos Example) (p. 82) and Output Model (Photos Example) (p. 83) in the Photos Example (p. 81)
- Input Model (News Article Example) (p. 85) and Output Model (News Article Example) (p. 87) in the News Article Example (p. 84)
- Input Model (Sales Invoice Example) (p. 89) and Output Model (Sales Invoice Example) (p. 91) in the Sales Invoice Example (p. 88)
- Input Model (Employee Record Example) (p. 94) and Output Model (Employee Record Example) (p. 96) in the Employee Record Example (p. 93)

To experiment with models in API Gateway, follow the instructions in Transform Response Payload (p. 41), specifically Step 1: Create Models (p. 43).

Mapping Templates

In API Gateway, a mapping template is used to transform some data from one format to another. You create and use input mapping templates and output mapping templates when you need to inform API Gateway about the schema of the data being sent from or returned to the caller, respectively. API Gateway uses the Velocity Template Language (VTL) and JSONPath expressions to define mapping templates.

For an example of an input mapping template, consider the example JSON data from the previous section. The following input mapping template makes no transform to the JSON data as API Gateway receives the JSON data from the caller:

```
#set($inputRoot = $input.path('$'))
{
  "department": "$inputRoot.department",
  "categories": [
#foreach($elem in $inputRoot.categories)
    "$elem"#if($foreach.hasNext),#end

#end
```

```
  ],
  "bins" : [
#foreach($elem in $inputRoot.bins)
    {
      "category" : "$elem.category",
      "type" : "$elem.type",
      "price" : $elem.price,
      "unit" : "$elem.unit",
      "quantity" : $elem.quantity
    }#if($foreach.hasNext),#end

#end
  ]
}
```

The preceding input mapping template is expressed as follows:

- Let the variable `$inputRoot` in the input mapping template represent the root object in the original JSON data.
- The values of the `department` object and `categories` and `bins` arrays in the input mapping template (represented by `$inputRoot.department`, `$inputRoot.categories`, and `$inputRoot.bins`) map to the corresponding values of the `department` object and `categories` and `bins` arrays in the root object in the original JSON data.
- In the input mapping template, each of the values in the `categories` array (represented by the first `$elem`), and each of the objects in the `bins` array (represented by the second `$elem`), map to the corresponding values in the `categories` array and objects in the `bins` array, respectively, within the root object in the original JSON data.
- For each of objects in the `bins` object, the values of the `category`, `type`, `price`, `unit`, and `quantity` objects in the input mapping template (represented by `$elem.category`, `$elem.type`, `$elem.price`, `$elem.unit`, and `$elem.quantity`, respectively) map to the corresponding values of the `category`, `type`, `price`, `unit`, and `quantity` objects in the original JSON data, respectively.

For an example of an output mapping template, first consider the following JSON data schema, which is based on the example JSON data from the previous section.

Note
None of the array and object names in this JSON data schema match the JSON data from the previous section:

```
{
  "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 transform the example JSON data from the previous section into this JSON data schema, you would use the following model:

```
{
  "$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 the preceding example, 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).

You would then use the following output mapping template, which is based on this model:

```
#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

#end
  ]
}
```

The preceding output mapping template is expressed as follows:

- Let the variable `$inputRoot` in the output mapping template represent the root object in the original JSON data from the previous section. Note the variables in the output mapping template map to the original JSON data, not the desired transformed JSON data schema.
- The `choices` array in the output mapping template maps to the `bins` array with the root object in the original JSON data (`$inputRoot.bins`).
- In the output mapping template, each of the objects in the `choices` array (represented by `$elem`) map to the corresponding objects in the `bins` array within the root object in the original JSON data.
- In the output mapping template, for each of objects in the `choices` object, the values of the `kind` and `available` objects (represented by `$elem.type` and `$elem.quantity`) map to 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.

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

- Input Mapping Template (Photos Example) (p. 82) and Output Mapping Template (Photos Example) (p. 84) in the Photos Example (p. 81)
- Input Mapping Template (News Article Example) (p. 86) and Output Mapping Template (News Article Example) (p. 87) in the News Article Example (p. 84)
- Input Mapping Template (Sales Invoice Example) (p. 90) and Output Mapping Template (Sales Invoice Example) (p. 92) in the Sales Invoice Example (p. 88)
- Input Mapping Template (Employee Record Example) (p. 95) and Output Mapping Template (Employee Record Example) (p. 97) in the Employee Record Example (p. 93)

To experiment with mapping templates in API Gateway, follow the instructions in Transform Response Payload (p. 41), specifically Step 5: Set Up and Test the Methods (p. 48).

Tasks for Models and Mapping Templates

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

- Create a Model (p. 79)
- View a List of Models (p. 80)
- Delete a Model (p. 80)

Create a Model in API Gateway

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

Topics

- Prerequisites (p. 80)
- Create a Model With the API Gateway Console (p. 80)

Prerequisites

- You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).

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 Set Up Request and Response Payload Mappings (p. 72).
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. 80)
- View a List of Models with the API Gateway Console (p. 80)

Prerequisites

- You must have at least one model in API Gateway. Follow the instructions in Create a Model (p. 79).

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 Set Up Request and Response Payload Mappings (p. 72).

Topics

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": "23456789@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" },
        "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 Set Up Request and Response Payload Mappings (p. 72).

Topics

Original Data (News Article Example)

The following is the original JSON data for the news article example:

```
{
  "count": 1,
  "items": [
    {
      "last_updated_date": "2015-04-24",
      "expire_date": "2016-04-25",
      "author_first_name": "John",
      "description": "Sample Description",
      "creation_date": "2015-04-20",
      "title": "Sample Title",
      "allow_comment": "1",
      "author": {
        "last_name": "Doe",
        "email": "johndoe@example.com",
        "first_name": "John"
      },
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "version": "1",
      "author_last_name": "Doe",
      "parent_id": 2345678901,
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

Input Model (News Article Example)

The following is the input model that corresponds to the original JSON data for the news article example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "NewsArticleInputModel",
  "type": "object",
  "properties": {
    "count": { "type": "integer" },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "last_updated_date": { "type": "string" },
          "expire_date": { "type": "string" },
          "author_first_name": { "type": "string" },
          "description": { "type": "string" },
          "creation_date": { "type": "string" },
          "title": { "type": "string" },
          "allow_comment": { "type": "string" },
          "author": {
            "type": "object",
            "properties": {
              "last_name": { "type": "string" },
              "email": { "type": "string" },
              "first_name": { "type": "string" }
            }
```

```
          },
          "body": { "type": "string" },
          "publish_date": { "type": "string" },
          "version": { "type": "string" },
          "author_last_name": { "type": "string" },
          "parent_id": { "type": "integer" },
          "article_url": { "type": "string" }
        }
      }
    },
    "version": { "type": "integer" }
  }
}
```

Input Mapping Template (News Article Example)

The following is the input mapping template that corresponds to the original JSON data for the news article example:

```
#set($inputRoot = $input.path('$'))
{
  "count": $inputRoot.count,
  "items": [
#foreach($elem in $inputRoot.items)
    {
      "last_updated_date": "$elem.last_updated_date",
      "expire_date": "$elem.expire_date",
      "author_first_name": "$elem.author_first_name",
      "description": "$elem.description",
      "creation_date": "$elem.creation_date",
      "title": "$elem.title",
      "allow_comment": "$elem.allow_comment",
      "author": {
        "last_name": "$elem.author.last_name",
        "email": "$elem.author.email",
        "first_name": "$elem.author.first_name"
      },
      "body": "$elem.body",
      "publish_date": "$elem.publish_date",
      "version": "$elem.version",
      "author_last_name": "$elem.author_last_name",
      "parent_id": $elem.parent_id,
      "article_url": "$elem.article_url"
    }#if($foreach.hasNext),#end

#end
  ],
  "version": $inputRoot.version
}
```

Transformed Data (News Article Example)

The following is one example of how the original news article example JSON data could be transformed for output:

```
{
  "count": 1,
  "items": [
    {
      "creation_date": "2015-04-20",
      "title": "Sample Title",
      "author": "John Doe",
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

Output Model (News Article Example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "NewsArticleOutputModel",
  "type": "object",
  "properties": {
    "count": { "type": "integer" },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "creation_date": { "type": "string" },
          "title": { "type": "string" },
          "author": { "type": "string" },
          "body": { "type": "string" },
          "publish_date": { "type": "string" },
          "article_url": { "type": "string" }
        }
      }
    },
    "version": { "type": "integer" }
  }
}
```

Output Mapping Template (News Article Example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "count": $inputRoot.count,
  "items": [
#foreach($elem in $inputRoot.items)
    {
      "creation_date": "$elem.creation_date",
```

```
      "title": "$elem.title",
      "author": "$elem.author.first_name $elem.author.last_name",
      "body": "$elem.body",
      "publish_date": "$elem.publish_date",
      "article_url": "$elem.article_url"
    }#if($foreach.hasNext),#end

#end
  ],
  "version": $inputRoot.version
}
```

Sales Invoice Example (API Gateway Models and Mapping Templates)

The following sections provide examples of models and mapping templates that could be used for a sample sales invoice API in API Gateway. For more information about models and mapping templates in API Gateway, see Set Up Request and Response Payload Mappings (p. 72).

Topics

Original Data (Sales Invoice Example)

The following is the original JSON data for the sales invoice example:

```
{
  "DueDate": "2013-02-15",
  "Balance": 1990.19,
  "DocNumber": "SAMP001",
  "Status": "Payable",
  "Line": [
    {
      "Description": "Sample Expense",
      "Amount": 500,
      "DetailType": "ExpenseDetail",
      "ExpenseDetail": {
        "Customer": {
          "value": "ABC123",
          "name": "Sample Customer"
        },
        "Ref": {
          "value": "DEF234",
          "name": "Sample Construction"
        },
        "Account": {
          "value": "EFG345",
```

```
          "name": "Fuel"
        },
        "LineStatus": "Billable"
      }
    }
  ],
  "Vendor": {
    "value": "GHI456",
    "name": "Sample Bank"
  },
  "APRef": {
    "value": "HIJ567",
    "name": "Accounts Payable"
  },
  "TotalAmt": 1990.19
}
```

Input Model (Sales Invoice Example)

The following is the input model that corresponds to the original JSON data for the sales invoice example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "InvoiceInputModel",
  "type": "object",
  "properties": {
    "DueDate": { "type": "string" },
    "Balance": { "type": "number" },
    "DocNumber": { "type": "string" },
    "Status": { "type": "string" },
    "Line": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "Description": { "type": "string" },
          "Amount": { "type": "integer" },
          "DetailType": { "type": "string" },
          "ExpenseDetail": {
            "type": "object",
            "properties": {
              "Customer": {
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "Ref": {
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "Account": {
```

```
                "type": "object",
                "properties": {
                  "value": { "type": "string" },
                  "name": { "type": "string" }
                }
              },
              "LineStatus": { "type": "string" }
            }
          }
        }
      }
    },
    "Vendor": {
      "type": "object",
      "properties": {
        "value": { "type": "string" },
        "name": { "type": "string" }
      }
    },
    "APRef": {
      "type": "object",
      "properties": {
        "value": { "type": "string" },
        "name": { "type": "string" }
      }
    },
    "TotalAmt": { "type": "number" }
  }
}
```

Input Mapping Template (Sales Invoice Example)

The following is the input mapping template that corresponds to the original JSON data for the sales invoice example:

```
#set($inputRoot = $input.path('$'))
{
  "DueDate": "$inputRoot.DueDate",
  "Balance": $inputRoot.Balance,
  "DocNumber": "$inputRoot.DocNumber",
  "Status": "$inputRoot.Status",
  "Line": [
#foreach($elem in $inputRoot.Line)
    {
      "Description": "$elem.Description",
      "Amount": $elem.Amount,
      "DetailType": "$elem.DetailType",
      "ExpenseDetail": {
        "Customer": {
          "value": "$elem.ExpenseDetail.Customer.value",
          "name": "$elem.ExpenseDetail.Customer.name"
        },
        "Ref": {
          "value": "$elem.ExpenseDetail.Ref.value",
          "name": "$elem.ExpenseDetail.Ref.name"
        },
```

```
        "Account": {
          "value": "$elem.ExpenseDetail.Account.value",
          "name": "$elem.ExpenseDetail.Account.name"
        },
        "LineStatus": "$elem.ExpenseDetail.LineStatus"
      }
    }#if($foreach.hasNext),#end

#end
  ],
  "Vendor": {
    "value": "$inputRoot.Vendor.value",
    "name": "$inputRoot.Vendor.name"
  },
  "APRef": {
    "value": "$inputRoot.APRef.value",
    "name": "$inputRoot.APRef.name"
  },
  "TotalAmt": $inputRoot.TotalAmt
}
```

Transformed Data (Sales Invoice Example)

The following is one example of how the original sales invoice example JSON data could be transformed for output:

```
{
  "DueDate": "2013-02-15",
  "Balance": 1990.19,
  "DocNumber": "SAMP001",
  "Status": "Payable",
  "Line": [
    {
      "Description": "Sample Expense",
      "Amount": 500,
      "DetailType": "ExpenseDetail",
      "Customer": "ABC123 (Sample Customer)",
      "Ref": "DEF234 (Sample Construction)",
      "Account": "EFG345 (Fuel)",
      "LineStatus": "Billable"
    }
  ],
  "TotalAmt": 1990.19
}
```

Output Model (Sales Invoice Example)

The following is the output model that corresponds to the transformed JSON data format:

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

```
    "Balance": { "type": "number" },
    "DocNumber": { "type": "string" },
    "Status": { "type": "string" },
    "Line": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "Description": { "type": "string" },
          "Amount": { "type": "integer" },
          "DetailType": { "type": "string" },
          "Customer": { "type": "string" },
          "Ref": { "type": "string" },
          "Account": { "type": "string" },
          "LineStatus": { "type": "string" }
        }
      }
    },
    "TotalAmt": { "type": "number" }
  }
}
```

Output Mapping Template (Sales Invoice Example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "DueDate": "$inputRoot.DueDate",
  "Balance": $inputRoot.Balance,
  "DocNumber": "$inputRoot.DocNumber",
  "Status": "$inputRoot.Status",
  "Line": [
#foreach($elem in $inputRoot.Line)
    {
      "Description": "$elem.Description",
      "Amount": $elem.Amount,
      "DetailType": "$elem.DetailType",
     "Customer": "$elem.ExpenseDetail.Customer.value ($elem.ExpenseDetail.Cus
tomer.name)",
      "Ref": "$elem.ExpenseDetail.Ref.value ($elem.ExpenseDetail.Ref.name)",
      "Account": "$elem.ExpenseDetail.Account.value ($elem.ExpenseDetail.Ac
count.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 Set Up Request and Response Payload Mappings (p. 72).

Topics

Original Data (Employee Record Example)

The following is the original JSON data for the employee record example:

```
{
  "QueryResponse": {
    "maxResults": "1",
    "startPosition": "1",
    "Employee": {
      "Organization": "false",
      "Title": "Mrs.",
      "GivenName": "Jane",
      "MiddleName": "Lane",
      "FamilyName": "Doe",
      "DisplayName": "Jane Lane Doe",
      "PrintOnCheckName": "Jane Lane Doe",
      "Active": "true",
      "PrimaryPhone": { "FreeFormNumber": "505.555.9999" },
      "PrimaryEmailAddr": { "Address": "janedoe@example.com" },
      "EmployeeType": "Regular",
      "status": "Synchronized",
      "Id": "ABC123",
      "SyncToken": "1",
      "MetaData": {
        "CreateTime": "2015-04-26T19:45:03Z",
        "LastUpdatedTime": "2015-04-27T21:48:23Z"
      },
      "PrimaryAddr": {
        "Line1": "123 Any Street",
        "City": "Any City",
        "CountrySubDivisionCode": "WA",
        "PostalCode": "01234"
      }
    }
  },
  "time": "2015-04-27T22:12:32.012Z"
}
```

Input Model (Employee Record Example)

The following is the input model that corresponds to the original JSON data for the employee record example:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "EmployeeInputModel",
  "type": "object",
  "properties": {
    "QueryResponse": {
      "type": "object",
      "properties": {
        "maxResults": { "type": "string" },
        "startPosition": { "type": "string" },
        "Employee": {
          "type": "object",
          "properties": {
            "Organization": { "type": "string" },
            "Title": { "type": "string" },
            "GivenName": { "type": "string" },
            "MiddleName": { "type": "string" },
            "FamilyName": { "type": "string" },
            "DisplayName": { "type": "string" },
            "PrintOnCheckName": { "type": "string" },
            "Active": { "type": "string" },
            "PrimaryPhone": {
              "type": "object",
              "properties": {
                "FreeFormNumber": { "type": "string" }
              }
            },
            "PrimaryEmailAddr": {
              "type": "object",
              "properties": {
                "Address": { "type": "string" }
              }
            },
            "EmployeeType": { "type": "string" },
            "status": { "type": "string" },
            "Id": { "type": "string" },
            "SyncToken": { "type": "string" },
            "MetaData": {
              "type": "object",
              "properties": {
                "CreateTime": { "type": "string" },
                "LastUpdatedTime": { "type": "string" }
              }
            },
            "PrimaryAddr": {
              "type": "object",
              "properties": {
                "Line1": { "type": "string" },
                "City": { "type": "string" },
                "CountrySubDivisionCode": { "type": "string" },
                "PostalCode": { "type": "string" }
              }
```

```
            }
          }
        }
      }
    },
    "time": { "type": "string" }
  }
}
```

Input Mapping Template (Employee Record Example)

The following is the input mapping template that corresponds to the original JSON data for the employee record example:

```
#set($inputRoot = $input.path('$'))
{
  "QueryResponse": {
    "maxResults": "$inputRoot.QueryResponse.maxResults",
    "startPosition": "$inputRoot.QueryResponse.startPosition",
    "Employee": {
      "Organization": "$inputRoot.QueryResponse.Employee.Organization",
      "Title": "$inputRoot.QueryResponse.Employee.Title",
      "GivenName": "$inputRoot.QueryResponse.Employee.GivenName",
      "MiddleName": "$inputRoot.QueryResponse.Employee.MiddleName",
      "FamilyName": "$inputRoot.QueryResponse.Employee.FamilyName",
      "DisplayName": "$inputRoot.QueryResponse.Employee.DisplayName",
     "PrintOnCheckName": "$inputRoot.QueryResponse.Employee.PrintOnCheckName",

      "Active": "$inputRoot.QueryResponse.Employee.Active",
      "PrimaryPhone": { "FreeFormNumber": "$inputRoot.QueryResponse.Employ
ee.PrimaryPhone.FreeFormNumber" },
      "PrimaryEmailAddr": { "Address": "$inputRoot.QueryResponse.Employ
ee.PrimaryEmailAddr.Address" },
      "EmployeeType": "$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": "$inputRoot.QueryResponse.Employee.MetaData.LastUp
datedTime"
      },
      "PrimaryAddr" : {
        "Line1": "$inputRoot.QueryResponse.Employee.PrimaryAddr.Line1",
        "City": "$inputRoot.QueryResponse.Employee.PrimaryAddr.City",
       "CountrySubDivisionCode": "$inputRoot.QueryResponse.Employee.PrimaryAd
dr.CountrySubDivisionCode",
       "PostalCode": "$inputRoot.QueryResponse.Employee.PrimaryAddr.PostalCode"

      }
    }
  },
  "time": "$inputRoot.time"
}
```

Transformed Data (Employee Record Example)

The following is one example of how the original employee record example JSON data could be transformed for output:

```
{
  "QueryResponse": {
    "maxResults": "1",
    "startPosition": "1",
    "Employees": [
      {
        "Title": "Mrs.",
        "GivenName": "Jane",
        "MiddleName": "Lane",
        "FamilyName": "Doe",
        "DisplayName": "Jane Lane Doe",
        "PrintOnCheckName": "Jane Lane Doe",
        "Active": "true",
        "PrimaryPhone": "505.555.9999",
        "Email": [
          {
            "type": "primary",
            "Address": "janedoe@example.com"
          }
        ],
        "EmployeeType": "Regular",
        "PrimaryAddr": {
          "Line1": "123 Any Street",
          "City": "Any City",
          "CountrySubDivisionCode": "WA",
          "PostalCode": "01234"
        }
      }
    ]
  },
  "time": "2015-04-27T22:12:32.012Z"
}
```

Output Model (Employee Record Example)

The following is the output model that corresponds to the transformed JSON data format:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "EmployeeOutputModel",
  "type": "object",
  "properties": {
    "QueryResponse": {
      "type": "object",
      "properties": {
        "maxResults": { "type": "string" },
        "startPosition": { "type": "string" },
        "Employees": {
          "type": "array",
          "items": {
            "type": "object",
```

```
            "properties": {
              "Title": { "type": "string" },
              "GivenName": { "type": "string" },
              "MiddleName": { "type": "string" },
              "FamilyName": { "type": "string" },
              "DisplayName": { "type": "string" },
              "PrintOnCheckName": { "type": "string" },
              "Active": { "type": "string" },
              "PrimaryPhone": { "type": "string" },
              "Email": {
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "type": { "type": "string" },
                    "Address": { "type": "string" }
                  }
                }
              },
              "EmployeeType": { "type": "string" },
              "PrimaryAddr": {
                "type": "object",
                "properties": {
                  "Line1": {"type": "string" },
                  "City": { "type": "string" },
                  "CountrySubDivisionCode": { "type": "string" },
                  "PostalCode": { "type": "string" }
                }
              }
            }
          }
        }
      }
    },
    "time": { "type": "string" }
  }
}
```

Output Mapping Template (Employee Record Example)

The following is the output mapping template that corresponds to the transformed JSON data format. The template variables here are based on the original, not transformed, JSON data format:

```
#set($inputRoot = $input.path('$'))
{
  "QueryResponse": {
    "maxResults": "$inputRoot.QueryResponse.maxResults",
    "startPosition": "$inputRoot.QueryResponse.startPosition",
    "Employees": [
      {
        "Title": "$inputRoot.QueryResponse.Employee.Title",
        "GivenName": "$inputRoot.QueryResponse.Employee.GivenName",
        "MiddleName": "$inputRoot.QueryResponse.Employee.MiddleName",
        "FamilyName": "$inputRoot.QueryResponse.Employee.FamilyName",
        "DisplayName": "$inputRoot.QueryResponse.Employee.DisplayName",
       "PrintOnCheckName": "$inputRoot.QueryResponse.Employee.PrintOnCheckName",
```

```
        "Active": "$inputRoot.QueryResponse.Employee.Active",
        "PrimaryPhone": "$inputRoot.QueryResponse.Employee.PrimaryPhone.Free
FormNumber",
        "Email" : [
          {
            "type": "primary",
            "Address": "$inputRoot.QueryResponse.Employee.PrimaryEmailAddr.Ad
dress"
          }
        ],
        "EmployeeType": "$inputRoot.QueryResponse.Employee.EmployeeType",
        "PrimaryAddr": {
          "Line1": "$inputRoot.QueryResponse.Employee.PrimaryAddr.Line1",
          "City": "$inputRoot.QueryResponse.Employee.PrimaryAddr.City",
         "CountrySubDivisionCode": "$inputRoot.QueryResponse.Employee.Primary
Addr.CountrySubDivisionCode",
          "PostalCode": "$inputRoot.QueryResponse.Employee.PrimaryAd
dr.PostalCode"
        }
      }
    ]
  },
  "time": "$inputRoot.time"
}
```

Amazon API Gateway API Request and Response Parameter-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. 102), `stage` (p. 106) or `util` (p. 107) 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. 244).

Topics

- Map Data to Integration Request Parameters (p. 98)
- Map Data to Method Response Headers (p. 100)
- Transform Request and Response Bodies (p. 101)

Map 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.`*`PARAM_NAME`*
Method request query string	`method.request.querystring.`*`PARAM_NAME`*
Method request header	`method.request.header.`*`PARAM_NAME`*
Method request body	`method.request.body`
Method request body (JsonPath)	`method.request.body.`*`JSONPath_EXPRESSION`*.
Stage variables	`stageVariables.`*`VARIABLE_NAME`*
Context variables	`context.`*`VARIABLE_NAME`* that must be one of the supported context variables (p. 102).
Static value	*`'STATIC_VALUE'`*. The *`STATIC_VALUE`* 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 `methodRequestHeadParam`, 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.meth
odRequestHeaderParam",
    "integration.request.querystring.integrationQueryParam" : "method.re
quest.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.pet
store.pets[0].name",

}
...
```

Map Data to Method Response Headers

Method response header parameters can be mapped from any integration response header or from the integration response body.

Method response header mapping expressions

Mapped Data Source	Mapping expression
Integration response header	`integration.response.header.`*`PARAM_NAME`*
Integration response body	`integration.response.body`
Integration response body (JsonPath)	`integration.response.body.`*`JSONPath_EXPRESSION`*
Stage variable	`stageVariables.`*`VARIABLE_NAME`*
Context variable	`context.`*`VARIABLE_NAME`* that must be one of the supported context variables (p. 102).
Static value	*`'STATIC_VALUE'`*. The *`STATIC_VALUE`* is a string literal and must be enclosed within a pair of single quotes.

Example data mapping from integration response in 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",

}
...
```

Transform Request and Response Bodies

Integration request and method response bodies can be transformed from the method request and integration response bodies, respectively, by using Mapping Templates (p. 76) written in Velocity Template Language (VTL). JSON data can be manipulated using VTL logic and JSONPath expressions, and additional data can be included from HTTP parameters, the calling context, and stage variables.

Select Mapping Templates

The request mapping template used to transform the method request body into the integration request body is selected by the value of the "Content-Type" header sent in the client request.

The response mapping template used to transform the integration response body into the method response body is selected by the value of the "Accept" header sent in the client request.

For example, if the client sends headers of `"Content-Type : application/xml"`, and `"Accept : application/json"`, the request template with the `application/xml` key will be used for the integration request, and the response template with the `application/json` key will be used for the method response.

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.

API Gateway API Request and Response Payload-Mapping Template Reference

Amazon API Gateway defines a set of variables 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

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.principalId`	The principal user identification associated with the token sent by the client.
`$context.authorizer.claims.`*`property`*	A property of the claims returned from the Amazon Cognito user pool after the method caller is successfully authenticated. **Note** Calling `$context.authorizer.claims` returns null.
`$context.httpMethod`	The HTTP method used. Valid values include: `DELETE`, `GET`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, and `PUT`.
`$context.identity.accountId`	The AWS account ID associated with the request.
`$context.identity.apiKey`	The API owner key associated with your API.
`$context.identity.caller`	The principal identifier of the caller making the request.
`$context.identity.cognitoAuthenticationProvider`	The Amazon Cognito authentication provider used by the caller making the request. Available only if the request was signed with Amazon Cognito credentials. For information related to this and the other Amazon Cognito `$context` variables, see Amazon Cognito Identity.
`$context.identity.cognitoAuthenticationType`	The Amazon Cognito authentication type of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.cognitoIdentityId`	The Amazon Cognito identity ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.cognitoIdentityPoolId`	The Amazon Cognito identity pool ID of the caller making the request. Available only if the request was signed with Amazon Cognito credentials.
`$context.identity.sourceIp`	The source IP address of the TCP connection making the request to API Gateway.

Parameter	Description
`$context.identity.user`	The principal identifier of the user making the request.
`$context.identity.userAgent`	The User Agent of the API caller.
`$context.identity.userArn`	The Amazon Resource Name (ARN) of the effective user identified after authentication.
`$context.requestId`	An automatically generated ID for the API call.
`$context.resourceId`	The identifier API Gateway assigns to your resource.
`$context.resourcePath`	The path to your resource. For more information, see Build an API Gateway API Step by Step (p. 14).
`$context.stage`	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 how to get context variables:

```
{
    "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"
}
```

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
`$input.body`	Returns the raw payload as a string.
`$input.json(x)`	This function evaluates a JSONPath expression and returns the results as a JSON string. For example, `$input.json('$.pets')` will return a JSON string representing the pets structure. For more information about JSONPath, see JSONPath or JSONPath for Java.
`$input.params()`	Returns a map of all the request parameters of your API call.
`$input.params(x)`	Returns the value of a method request parameter from the path, query string, or header value (in that order) given a parameter name string x.
`$input.path(x)`	Takes a JSONPath expression string (x) and returns an object representation of the result. This allows you to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL). For example, `$input.path('$.pets').size()` 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 Set Up Request and Response Payload Mappings (p. 72)

Accessing the $stageVariables Variable

The syntax for inserting a stage variable looks like this: `$stageVariables`.

$stageVariables Reference

Syntax	Description
`$stageVariables.<variable_name>`	*<variable_name>* represents a stage variable name.

Syntax	Description
`$stageVariables['<variable_name>']`	*<variable_name>* represents any stage variable name.
`${stageVariables['<variable_name>']}`	*<variable_name>* 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
`$util.escapeJavaScript()`	Escapes the characters in a string using JavaScript string rules. **Note** This function will turn any regular single quotes (`'`) into escaped ones (`\'`). However, the escaped single quotes are not valid in JSON. Thus, when the output from this function is used in a JSON property, you must turn any escaped single quotes (`\'`) back to regular single quotes (`'`). This is shown in the following example: `$util.escapeJavaScript(data).replaceAll("\'","'")`

Function	Description

`$util.parseJson()`	Takes "stringified" JSON and returns an object representation of the result. You can use the result from this function to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL). For example, if you have the following payload:
	`{"errorMessage":"{\"key1\":\"var1\",\"key2\":{\"arr\":[1,2,3]}}"}`
	and use the following mapping template
	`#set ($errorMessageObj = $util.parseJson($input.path('$.errorMessage')))`
	`{`
	`  "errorMessageObjKey2ArrVal" : $errorMessageObj.key2.arr[0]`
	`}`
	You will get the following output:
	`{`
	`  "errorMessageObjKey2ArrVal" : 1`
	`}`

`$util.urlEncode()`	Converts a string into "application/x-www-form-urlencoded" format.
`$util.urlDecode()`	Decodes an "application/x-www-form-urlencoded" string.
`$util.base64Encode()`	Encodes the data into a base64-encoded string.
`$util.base64Decode()`	Decodes the data from a base64-encoded string.

Integration Passthrough Behaviors

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 back end without transformation. The process is known as integration passthrough. 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. 67), 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
None (default to `application/json`	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
`application/json`	The request payload is transformed using the template.	The request payload is transformed using the template.	The request payload is transformed using the template.
`application/xml`	The request payload is not transformed and is sent to the back end 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 back end 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 back end 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.

Import and Export API Gateway API with Swagger Definition Files

As an alternative to using the Amazon API Gateway console to create and update your API, you can use the API Gateway Import API feature to upload API definitions into API Gateway from external API definition files, such as those using the Swagger specification with the API Gateway extensions (p. 115).

After an API is created and configured in API Gateway, you can download it as a Swagger definition file using the Amazon API Gateway Export API. The API Gateway console has enabled this feature for you to export an API using intuitive visual interfaces.

Topics

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 definition as the payload, or you can update an existing API by using a PUT request that contains a 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 in the request URL using a `mode` query parameter.

> **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. The option is available as an item in the **Actions** drop-down menu. For an example of using the Import API feature from the API Gateway console, see Build and Test an API Gateway API from an Example (p. 6).

Use the Import API to Create a New API

To use the Import API feature to create a new API, POST your API definition file to `https://apigateway.<region>.amazonaws.com/restapis?mode=import`. This request results in a new *RestApi*, along with *Resources*, *Models*, and other items defined in the definition file.

The following code snippet shows an example of the POST request with the payload of a JSON-formatted Swagger definition:

```
POST /restapis?mode=import
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
Content-Length: ...

Swagger API definition in JSON (p. 155)
```

Use the Import API to Update an Existing API

You can use the Import API feature to update an existing API when there are aspects of that API you would like to preserve, such as stages and stage variables, or references to the API from API Keys.

An API update can occur in two modes: merge or overwrite. 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.

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. 155)
```

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. 155)
```

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.

Swagger `basePath`

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.

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 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.

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 extension in JSON format:

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

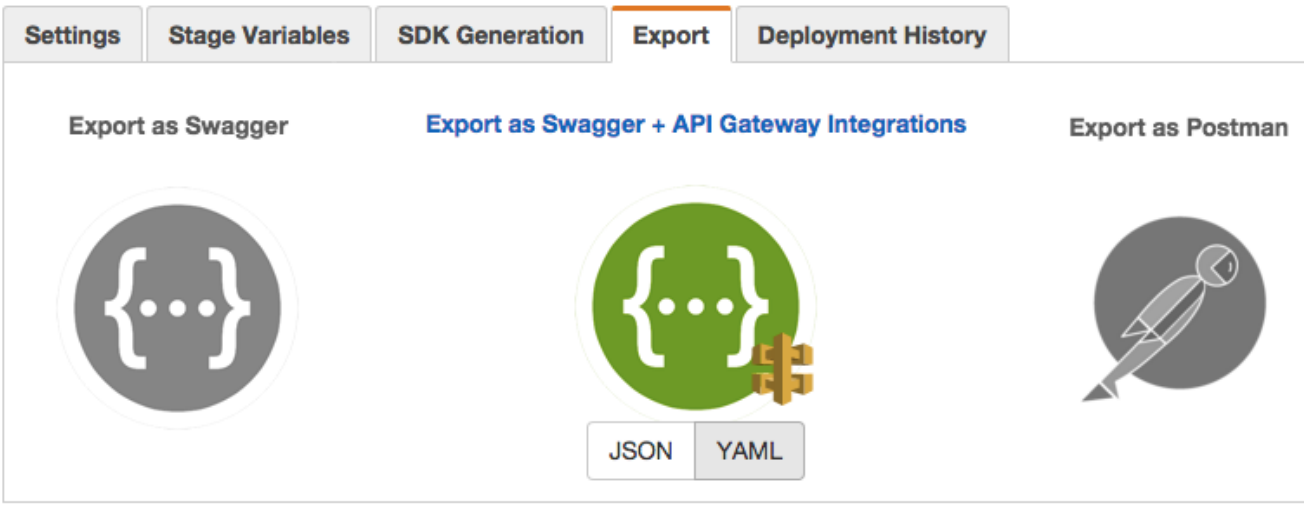
Download API Swagger Definition with API Gateway Integration in YAML

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=integ
ration
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

Export API 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.



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. 113).

Topics

x-amazon-apigateway-authorizer Object

Defines 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 Operation object.

Properties

Property Name	Type	Description
type	`string`	The type of the authorizer. This is a required property and the value must be "token".
authorizerUri	`string`	The Uniform Resource Identifier (URI) of the authorizer (a Lambda function). For example, `"arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:account-id:function:auth_function_name/invocations"`
authorizerCredentials	`string`	Credentials required for the authorizer, if any, in the form of an ARN of an IAM execution role. For example, "arn:aws:iam::*account-id*:*IAM_role*".
identityValidationExpression	`string`	A regular expression for validating the incoming identity. For example, "^x-[a-z]+".
authorizerResultTtlInSeconds	`string`	The number of seconds during which the resulting IAM policy is cached.

x-amazon-apigateway-authorizer Example

The following Swagger security definitions example specifies a custom authorizer named `test-authorizer`.

```
  "securityDefinitions" : {
    "test-authorizer" : {
      "type" : "apiKey",                          // Required and the value must
 be "apiKey" for an API Gateway API.
      "name" : "Authorization",                    // The source header name
identifying this authorizer.
      "in" : "header",                            // Required and the value must
 be "header" for an AAPI Gateway API.
     "x-amazon-apigateway-authtype" : "oauth2", // Specifies the authorization
 mechanism for the client.
      "x-amazon-apigateway-authorizer" : {       // An API Gateway custom au
thorizer 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/invoca
tions",
        "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 custom 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"
       }
     }
   }
```

x-amazon-apigateway-authtype Property

Specify the type of a custom authorizer. It is parsed by the API Gateway API import and export features.

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 custom authorizer using OAuth 2.

```
   "cust-authorizer" : {
     "type" : "...", // required
     "name" : "...", // name of the identity source header
     "in" : "header", // must be header
    "x-amazon-apigateway-authtype" : "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"
}
```

x-amazon-apigateway-integration Object

Specifies details of the back-end 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
type	`string`	The type of integration with the specified back end. The valid value is `http` (for integration with an HTTP back end) or `aws` (for integration with AWS Lambda functions or other AWS services, such as DynamoDB, SNS or SQS).
uri	`string`	The endpoint URI of the back end. For integrations of the `aws` type, this is an ARN value. For the HTTP integration, this is the URL of the HTTP endpoint including the `https` or `http` scheme.
httpMethod	`string`	The HTTP method used in the integration request. For Lambda function invocations, the value must be `POST`.

Property Name	Type	Description
credentials	`string`	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.
requestTemplates	x-amazon-apigateway-integration.requestTemplates (p. 120)	Mapping templates for a request payload of specified MIME types.
requestParameters	x-amazon-apigateway-integration.requestParameters (p. 121)	Specifies mappings from method request parameters to integration request parameters. Supported request parameters are `querystring`, `path`, `header`, and `body`.
cacheNamespace	`string`	An API-specific tag group of related cached parameters.
cacheKeyParameters	An array of `string`	A list of request parameters whose values are to be cached.
responses	x-amazon-apigateway-integration.responses (p. 122)	Defines the method's responses and specifies desired parameter mappings or payload mappings from integration responses to method responses.

x-amazon-apigateway-integration Example

The following example integrates an API's `POST` method with a Lambda function in the back end. 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/func
tions/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.querys
tring.vendor"
    },
    "cacheNamespace" : "cache namespace",
    "cacheKeyParameters" : [],
    "responses" : {
        "2\\d{2}" : {
            "statusCode" : "200",
            "responseParameters" : {
              "method.response.header.requestId" : "integration.response.head
er.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.re
direct.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
MIME type	`string`	An example of the MIME type is `application/json`. For information about creating a mapping template, see Mapping Templates (p. 76).

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.ver
sion",
```

```
    "integration.request.header.x-userid" : "method.request.header.x-user-id",

    "integration.request.path.op" : "method.request.path.service"
},
```

x-amazon-apigateway-integration.responses Object

Defines the method's responses and specifies parameter mappings or payload mappings from integration responses to method responses.

Properties

Property Name	Type	Description
Response status pattern	x-amazon-apigateway-integration.response (p. 123)	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 `errorMessage` field of the error information object returned by AWS Lambda as a failure response body when the Lambda function execution throws an exception. **Note** The *Response status pattern* property name refers to a response status code or regular expression describing a group of response status codes. It does not correspond to any identifier of an IntegrationResponse resource in the API Gateway REST API.

`x-amazon-apigateway-integration.responses` Example

The following example shows a list of responses from `2xx` and `302` responses. For the `2xx` response, the method response is mapped from the integration response's payload of the `application/json` or `application/xml` MIME type. This response uses the supplied mapping templates. For the `302` response, the method response returns a `Location` header whose value is derived from the `redirect.url` property on the integration response's payload.

```
"responses" : {
    "2\\d{2}" : {
        "statusCode" : "200",
        "responseTemplates" : {
```

```
            "application/json" : "#set ($root=$input.path('$')) { \"stage\":
\"$root.name\", \"user-id\": \"$root.key\" }",
            "application/xml" : "#set ($root=$input.path('$'))
<stage>$root.name</stage> "
        }
    },
    "302" : {
        "statusCode" : "302",
        "responseParameters" : {
          "method.response.header.Location": "integration.response.body.redir
ect.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 (p. 124)	Specifies MIME type-specific mapping templates for the response's payload.
responseParameters	x-amazon-apigateway-integration.responseParameters (p. 124)	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.

`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 back end. 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.redir
ect.url"
    }
}
```

x-amazon-apigateway-integration.responseTemplates Object

Specifies mapping templates for a response payload of the specified MIME types.

Properties

Property Name	Type	Description
MIME type	string	Specifies a mapping template to transform the integration response body to the method response body for a given MIME type. For information about creating a mapping template, see Mapping Templates (p. 76). An example of the *MIME type* is application/json.

x-amazon-apigateway-integration.responseTemplate Example

The following example sets mapping templates for a request payload of the application/json and application/xml MIME types.

```
"responseTemplates" : {
    "application/json" : "#set ($root=$input.path('$')) { \"stage\":
\"$root.name\", \"user-id\": \"$root.key\" }",
   "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage>
 "
}
```

x-amazon-apigateway-integration.responseParameters Object

Specifies mappings from integration method response parameters to method response parameters. 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.<param-name>`	`string`	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"
}
```

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 A Sample Amazon S3 Proxy API in Swagger with API Gateway Extensions (p. 134). To do so, copy the Swagger definition and paste it into a file. Use the API Gateway Swagger Importer. For more information, see Getting Started with the API Gateway Swagger Importer.

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 http://aws.amazon.com/ and choose **Create an AWS Account**.
2. Follow the online instructions.

To allow the API to invoke the 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.

Create an IAM Role and Policy for the API to Access Amazon S3

For read-only operations, including `Get*` and `List*` actions in Amazon S3, you can use the AmazonS3ReadOnlyAccess policy provided by the IAM , whose ARN is `arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess`:

The AmazonS3ReadOnlyAccess Policy

```
{
    "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.

To allow Amazon S3 buckets and objects to be updated, you can use a custom policy for any of the Amazon S3 `Put*` actions.

An Amazon S3 Put-only Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:Put*",
            "Resource": "*"
        }
    ]
```

```
}
```

You can attach a read-only and a put-only policy to an IAM role if your API works with Amazon S3 `Get*`, `List*` and `Put*` actions only.

To invoke the Amazon S3 `Post` actions, you must include `s3:Post*` action in an Allow policy document. For a complete list of Amazon S3 actions, see Specifying Amazon S3 Permissions in a Policy.

For an API to create, view, update, and delete buckets and objects in Amazon S3, you can attach a single full-access policy. For this, you can use the AmazonS3FullAccess policy, which is provided by the IAM console and whose ARN is arn:aws:iam::aws:policy/AmazonS3FullAccess.

The AmazonS3FullAccess Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": s3:*,
            "Resource": "*"
        }
    ]
}
```

This policy covers all actions on any resources in Amazon S3. Using the IAM role and policies ensures that API Gateway can call the specifically allowed Amazon S3 actions on the specified Amazon S3 resources.

After you have decided which IAM policy documents to use, create an IAM role. Attach the policies to the role. The resulting IAM role must contain the following trust policy for the attached policies to be applied on API Gateway.

```
{
  "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 for Amazon S3 Features

The following procedure describes how to use the API Gateway console to create an API that exposes the Amazon S3 service features.

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

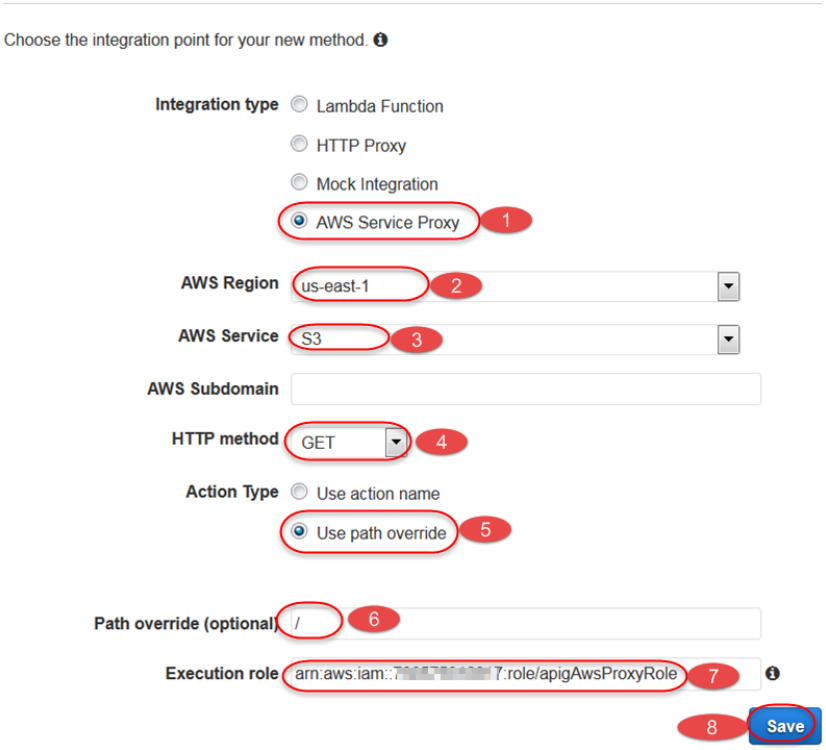
1. In the API Gateway console, create an API named **MyS3**. This API's root resource (***/***) represents the Amazon S3 service. Later, we will expose the GET method to list the Amazon S3 buckets of the caller.
2. For the API's root resource, create a child resource named **Folder**, setting the required **Resource Path** as **/{folder}**. The **folder** path parameter enables the client to specify a bucket name in the URL when the client calls the API to work with the bucket. Later, we will expose the DELETE, GET, and PUT methods on this **Folder** resource to work with an Amazon S3 bucket in the back end. We will also declare a **bucket** path parameter for the back-end URL and specify a mapping expression to translate **folder** to **bucket**.
3. For the API's **Folder** resource, identified by the **/{folder}** resource path, create an **Item** child resource. Set the required **Resource Path** as **/{item}**. The **folder** and **item** path parameters enable the client to specify, in the request's URL, an object name in a given **folder** when the client calls the API to work with the object. Later, we will expose the DELETE, HEAD, GET and PUT methods on this **Item** resource. We will also declare **bucket** and **object** path parameters for the back-end URL and specify mapping expressions to translate **folder** and **item** to **bucket** and **object**, respectively.

Expose a GET Method on an API Root as Get Service Action in Amazon S3

Use **Create Method** in the API Gateway console to create a GET method for the API's root resource, (***/***). In the **Set up** pane for the method, configure the GET method to integrate with the GET Service action in Amazon S3, as follows.

To set up the newly created GET method on the API root

1. For the **Integration type**, choose **AWS Service Proxy**.
2. From the list, choose an **AWS Region**.
3. From **AWS Service**, choose **S3**.
4. From **HTTP method**, choose **GET**.
5. For **Action Type**, choose **Use path override**.
6. (Optional) In **Path override** type ***/***.
7. Copy the previously created IAM role's ARN (from the IAM console) and paste it into **Execution role**.
8. Choose **Save** to finish setting up this method.



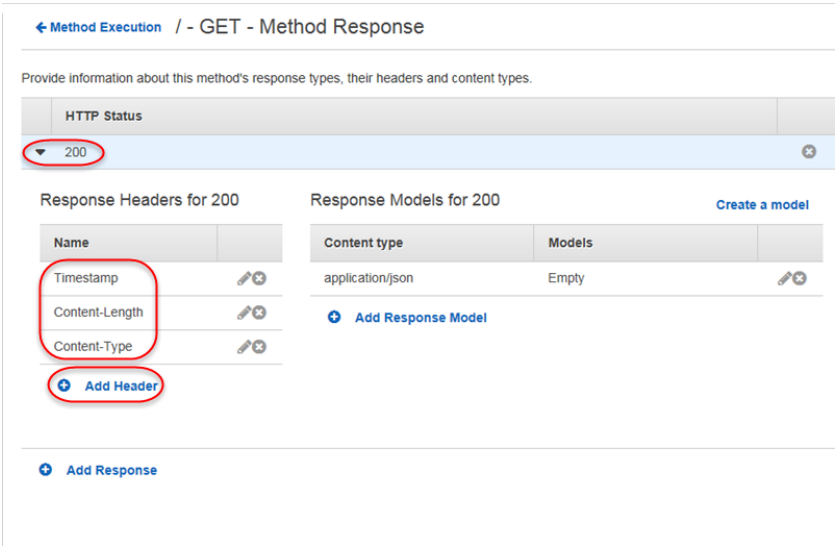
Note
After the method is set up, you can modify these settings in the method's **Integration Request** page.

By default, API Gateway assumes the request and response payload to be of the "application/json" type. However, Amazon S3 returns results in an XML-formatted response payload. This means that you must override the default Content-Type header value of the method response with the Content-Type header value from the integration response. Otherwise, the client will see "application/json" in the response Content-Type header when the response body is an XML string.

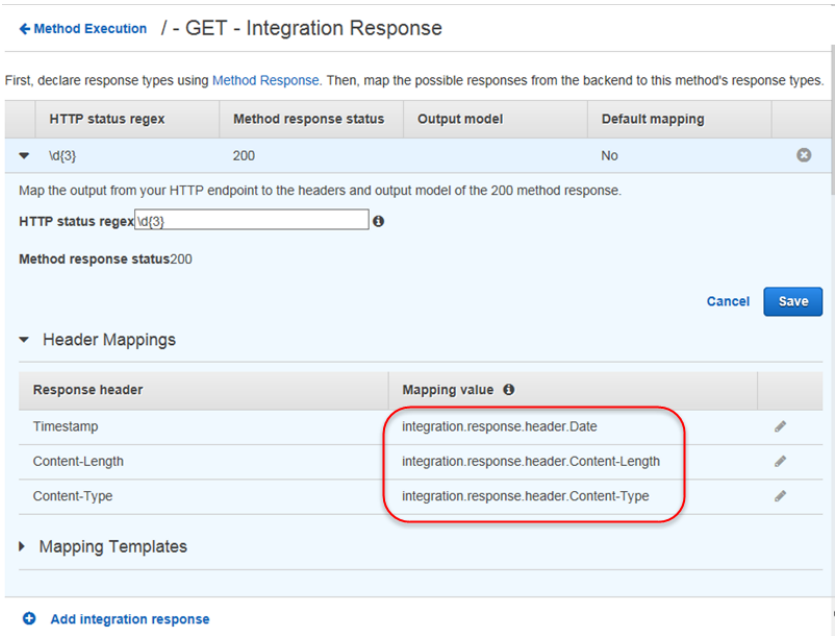
To translate the integration response header to the method response header, use response header mappings. The process involves declaring the Content-Type header explicitly for the method response and specifying a header mapping expression for the integration response to pass the back-end Content-Type header value to the front-end Content-Type header value.

To set up header mappings to return an integration response Content-Type header

1. In the API Gateway console, choose **Method Response**. Add the **Content-Type** header.

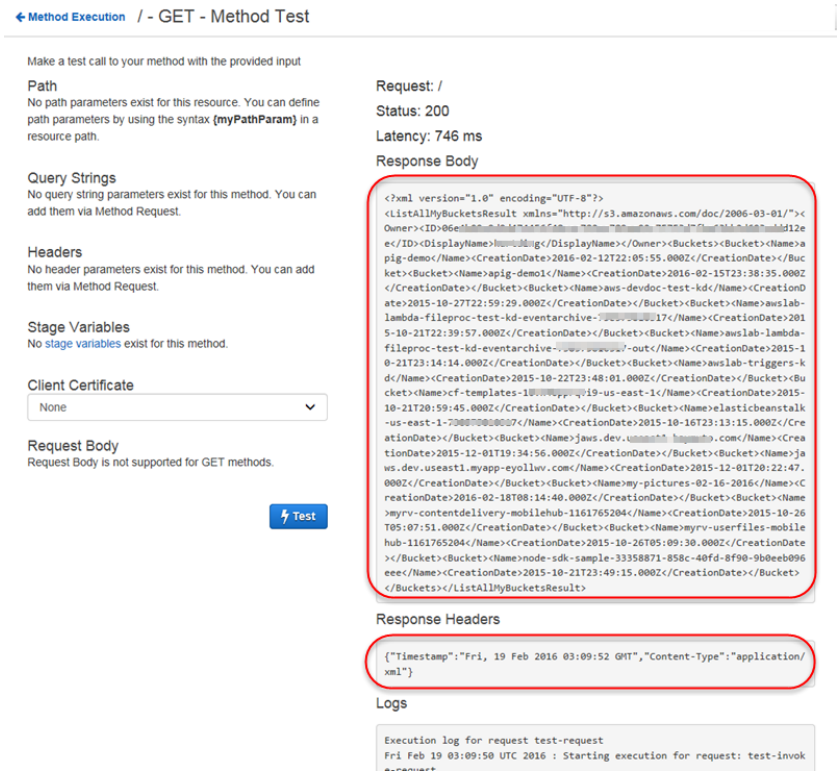


2. In **Integration Response**, for **Content-Type**, type `integration.response.header.Content-Type` for the method response.



Test the GET method on the API root resource

- In **Method Execution**, choose **Test**. An example result is shown in the following figure.



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.

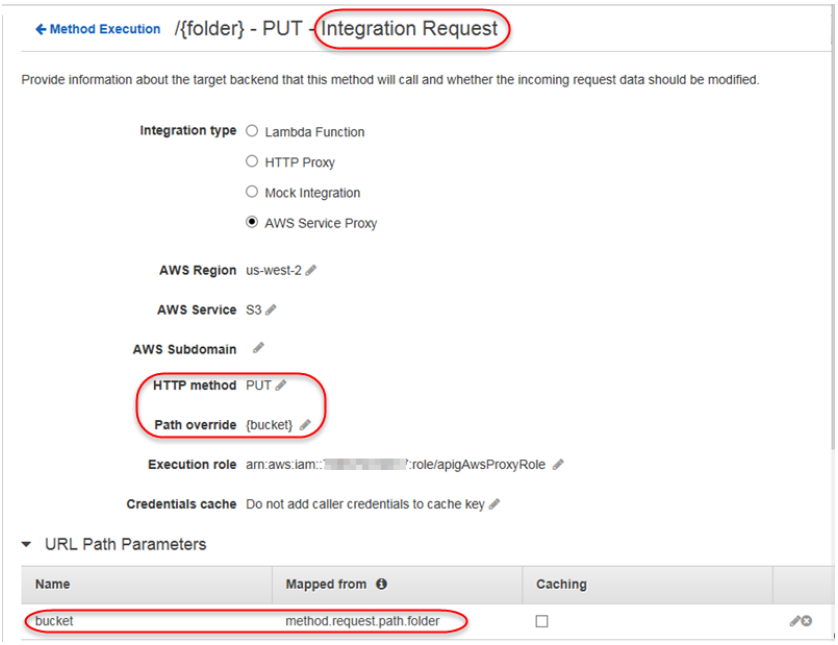
Expose Methods on an API Folder Resource as Bucket Actions in Amazon S3

You can apply a similar procedure to expose methods on the Folder (**/{folder}**) resources, with minor modifications that include using different HTTP verbs and setting up path parameter mappings between the method request and integration request. To illustrate, we expose the GET, PUT, and DELETE methods for listing objects in a bucket, creating a new bucket, and deleting an existing bucket, respectively. As an example, we will cover the PUT method setup in detail.

To expose methods on a folder resource

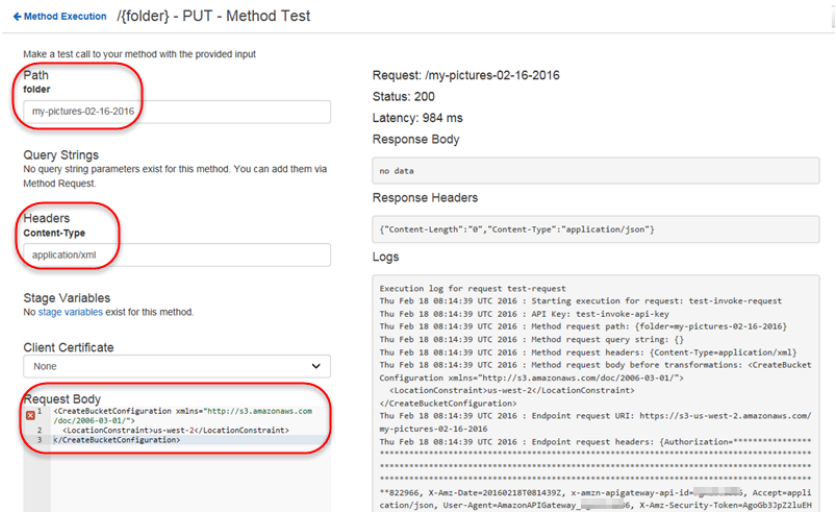
1. On the **/{folder}** resource under the root, choose **Create Method** to create a PUT method.
2. Follow the setup steps for the GET method on the root resource explained in the previous procedure, except that you will specify **PUT** for **HTTP method** and **`/{bucket}`** for **Path override**.
3. Set up the mappings for the Content-Type and possibly other headers from method request to integration request and from integration response to method response. The instructions are the same or similar to the header mapping from the integration response to the method response for the GET method on the API's root resource. You must add the mapping for the Content-Type header from the method request to integration because you must supply an XML payload in the PUT request; this mapping overrides the default Content-Type header value (i.e., application/json) to reflect the actual payload content type.

4. In **Integration Request**, expand the **URL Path Parameters** section. Add the path parameter name, for example, **bucket**, as specified in **Path override**. Type a path-mapping expression, namely **method.request.path.folder**, to map the front-end path parameter (**folder**) for the method request to the back-end path parameter (**bucket**) for the integration request.



5. Choose **Test** from the **Method Execution** pane to test this PUT method. In **folder**, type a bucket name, and then in **Content-Type**, type **`application/xml`**. In **Request Body**, provide the bucket region as the location constraint; it is declared in an XML fragment as the payload of the request.

```
<CreateBucketConfiguration>

<LocationConstraint>us-west-2</LocationConstraint>

</CreateBucketConfiguration>
```



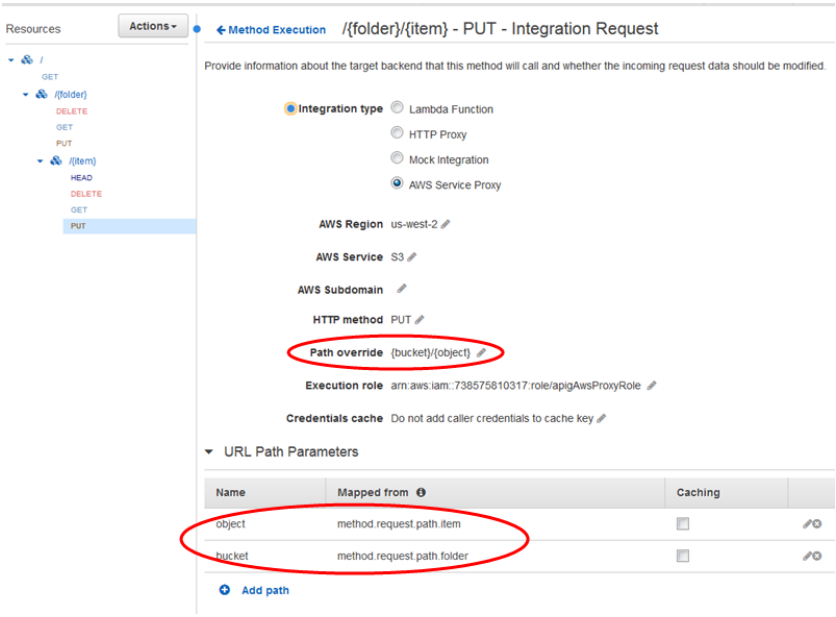
6. Repeat the preceding steps to create and configure the GET and DELETE method on the API's **/{folder}** resource.

Expose Methods on an API Item in a Folder as Actions on an Amazon S3 Object in a Bucket

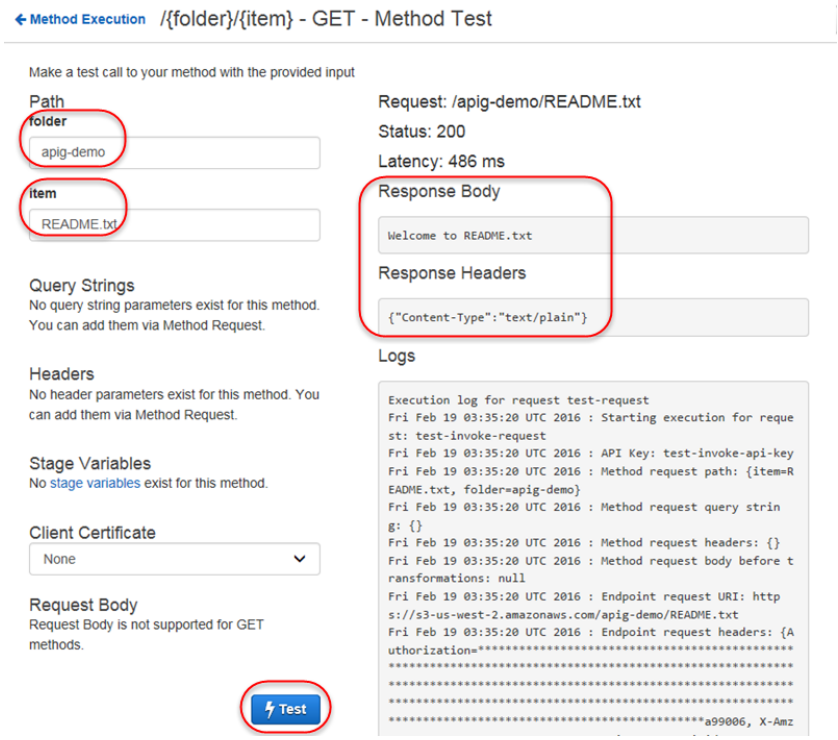
The following procedure describes how to expose methods on an API item in a folder as an action on an Amazon S3 object in a bucket.

To expose methods on an item resource

1. Repeat the preceding steps to create and configure the DELETE, HEAD, GET and PUT methods on the **/{folder}/{item}** resource. The following image shows integration settings for the PUT method. The settings for the other methods are similar.



2. To test the GET method on a Folder/Item resource using the API Gateway console, choose **Test** in the **Method Execution** page. The following image shows the result of an example test, where the README.txt file in the apig-demo bucket in Amazon S3 contains a string of plain text ("Welcome to README.txt").



A Sample Amazon S3 Proxy API in Swagger with API Gateway Extensions

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-02-19T04:30:12Z",
    "title": "MyS3"
  },
  "host": "1234567890.execute-api.us-east-1.amazonaws.com",
  "basePath": "/S3",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [],
```

```
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Timestamp": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Content-Length": "integration.re
sponse.header.Content-Length",
                "method.response.header.Timestamp": "integration.response.head
er.Date"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path//",
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Date": "integration.response.head
er.Date",
                "method.response.header.Content-Length": "integration.re
sponse.header.content-length"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "httpMethod": "GET",
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Content-Length": "integration.re
sponse.header.Content-Length"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "httpMethod": "PUT",
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder",
            "integration.request.header.Content-Type": "method.request.head
er.Content-Type"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
              "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
              "method.response.header.Date": "integration.response.header.Date"

              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "httpMethod": "DELETE",
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.content-type": "integration.response.head
er.content-type",
              "method.response.header.Content-Type": "integration.response.head
er.Content-Type"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "httpMethod": "GET",
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Content-Length": "integration.re
sponse.header.Content-Length"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "httpMethod": "HEAD",
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "\\d{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Content-Length": "integration.re
sponse.header.Content-Length"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "httpMethod": "PUT",
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.header.content-type": "method.request.head
er.Content-Type",
            "integration.request.path.bucket": "method.request.path.folder",
            "integration.request.header.Content-Type": "method.request.head
er.Content-Type"
          },
          "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"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "d\\{3}": {
              "statusCode": "200",
              "responseParameters": {
               "method.response.header.Content-Type": "integration.response.head
er.Content-Type",
                "method.response.header.Content-Length": "integration.re
sponse.header.Content-Length"
              },
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "httpMethod": "DELETE",
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
```

```
    "Empty": {
      "type": "object"
    }
  }
}
```

Create an API Gateway API as an AWS Lambda Proxy

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.

If your API makes only synchronous calls to Lambda functions in the back end, you should use the **Lambda Function** integration type. For instructions, see Call Lambda Functions Synchronously (p. 22).

If your API makes asynchronous calls to Lambda functions, you must use the **AWS Service Proxy** integration type described in this section. The instructions apply to requests for synchronous Lambda function invocations as well. For the asynchronous invocation, 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.

The API 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 back-end Lambda function through three query string parameters (`operand1`, `operand2`, and `operator`). These are mapped 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, which is then passed through to the integration request. Alternatively, the API 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. Parameter mappings and mapping templates are used to translate the method request data into the Lambda function input and to translate the output from the integration responses to the method response.

This section provides more detailed discussions for the following tasks:

- 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.
- Expose POST on the `/calc` resource to invoke the Lambda function, supplying the input in the payload.
- Expose GET on the `/calc/{operand1}/{operand2}/{operator}` resource to invoke the Lambda function, specifying the input in the path parameters.

You can import the sample API as a Lambda proxy from the Swagger Definitions of a Sample API as Lambda Proxy (p. 155). To do so, copy the Swagger definition, paste it into a file, and use the API Gateway Swagger Importer. For more information, see Getting Started with the API Gateway Swagger Importer.

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 http://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

The API will use the InvokeFunction action to call a Lambda function. 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 Back End

Copy the following Lambda function and paste it into the code editor in the Lambda console.

```
exports.handler = function(event, context) {
    //console.log('Received event:', JSON.stringify(event, null, 2));
    var res = {};
    res.a = event.a;
    res.b = event.b;
    res.op = event.op;

    switch(event.op)
    {
        case "+":
            res.c = Number(event.a) + Number(event.b);
            break;
        case "-":
            res.c = Number(event.a) - Number(event.b);
            break;
        case "*":
            res.c = Number(event.a) * Number(event.b);
            break;
        case "/":
            res.c = Number(event.b)===0 ? NaN : Number(event.a) / Num
ber(event.b);
```

```
            break;
        default:
            res.c = "Invalid op";
    }
    context.succeed(res);
};
```

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, which is created next.

Create API Resources for the Lambda Function

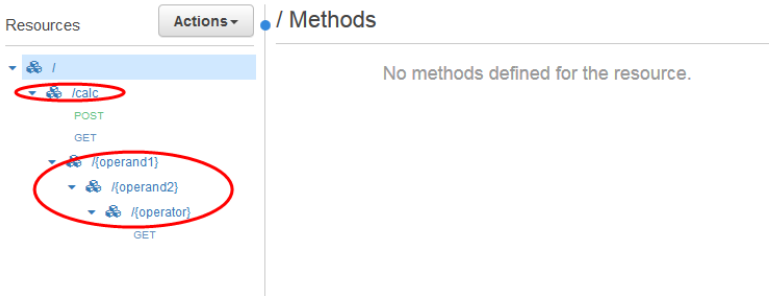
The following procedure describes how to create API resources for the Lambda function.

To create API resources for Lambda functions

1. In the API Gateway console, create an API named **LambdaGate**. You can create child resources to represent different Lambda functions; in the following, you will create a single child resource of the API root.
2. For the simple calculator function you created, create the **/calc** resource off the API's root. You will expose the GET and POST methods on this resource for the client to invoke the back-end Lambda function, supplying 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.

   You will also create the **/calc/{operand1}/{operand2}/{operator}** to expose the GET method to invoke the Lambda function and to supply the required input as the three path parameters (**operand1**, **operand2**, and **operator**).

   We will show how to apply API Gateway request and response data mapping to normalize the input to the back end Lambda function.



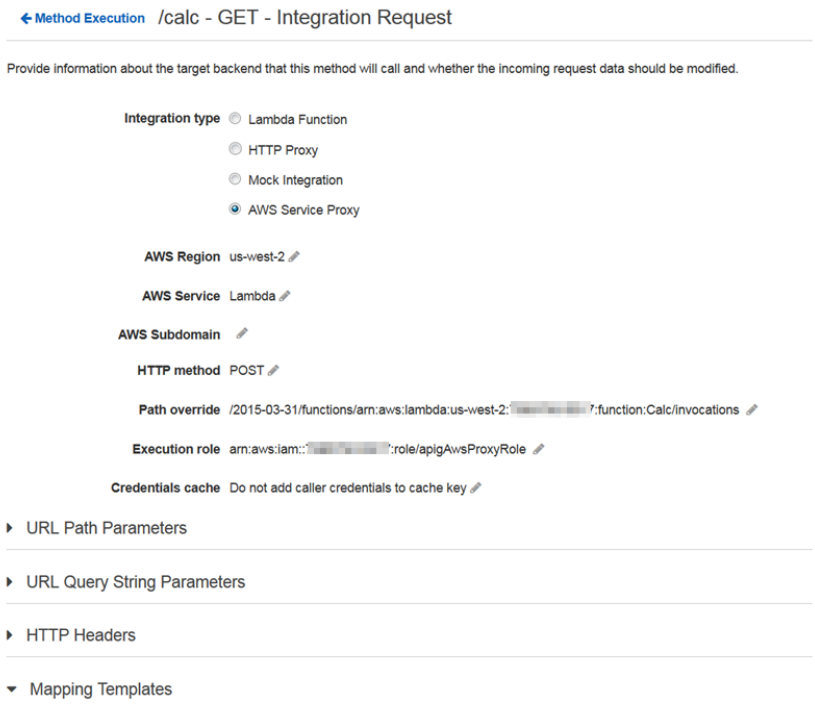
Create a GET Method with Query Strings to Call the Lambda Function

Use the following steps to expose a GET method with query strings to call a Lambda function.

To set up the GET method with query strings to invoke the Lambda function

1. Choose **Create Method** in the API Gateway console to create a GET method for the API's **/calc** resource.

   In the method's **Set up** pane, configure the method with the following settings.



   You must use the POST method for the integration request when calling a Lambda function, although you can use any other HTTP verbs for the method request.

   The **Path override** value must the URL path of the Lambda Invoke action. The path is of the following format:

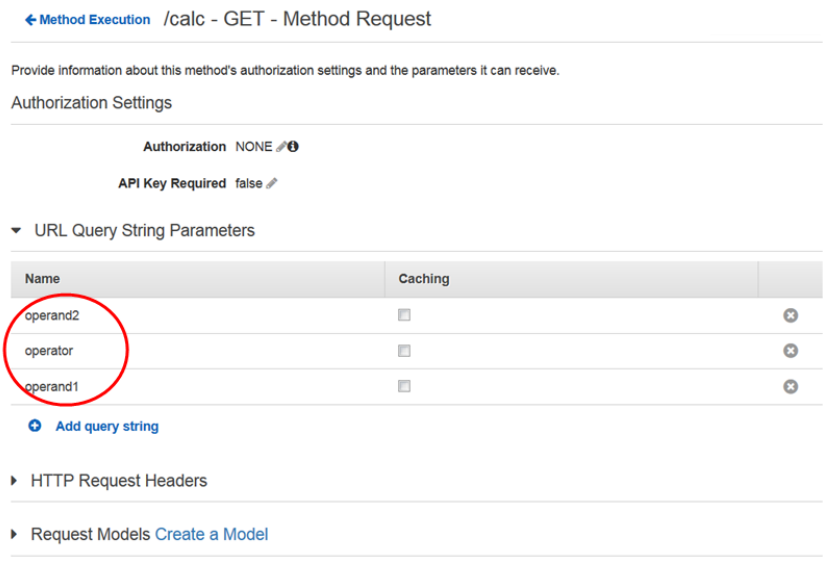
```
/2015-03-31/functions/FunctionName/invocations?Qualifier=version
```

where *FunctionName* is the ARN of the Lambda function to be invoked. The optional `Qualifier` query string can be used to select a version of the function. If it not specified, the `$LATEST` version will be used.

You can also add 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. For the example shown here, this header has the default value.

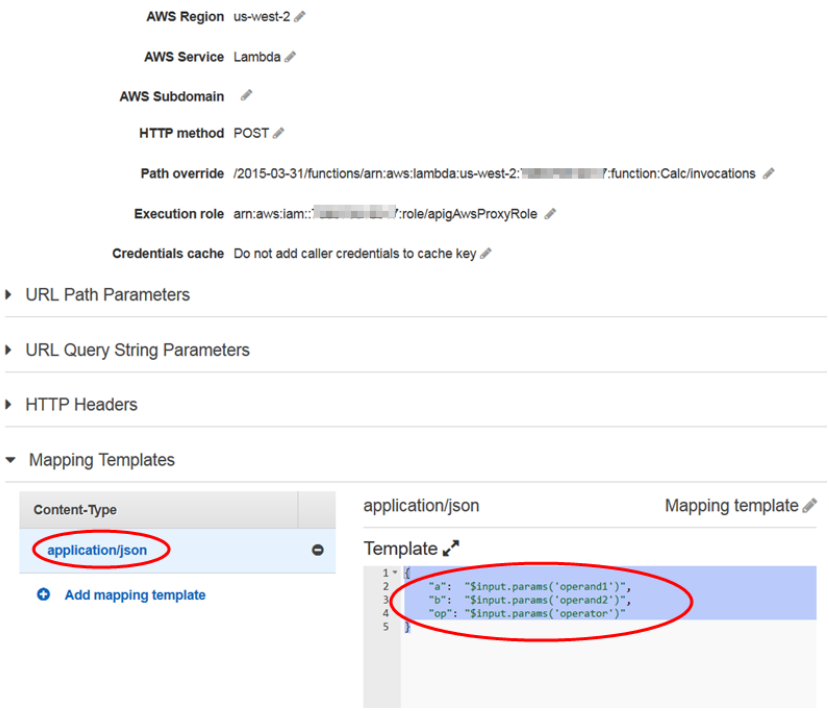
We will come back to setting up **Mapping Templates** after setting up the query string parameters to hold the input data for the Lambda function.

2. In **Method Request** for the **GET** method on **/calc**, expand the **URL Query String Parameters** section. Choose **Add query string** to add the **operand1**, **operand2**, and **operator** query string parameters.



3. Go back to **Integration Request**. Expand the **Mapping Templates** section. If necessary, in **Content-Type**, under **application/json**, choose **Add mapping template**. Type the following 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 back-end Lambda function. The transformed JSON object will be included as the integration request payload.

4. 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.

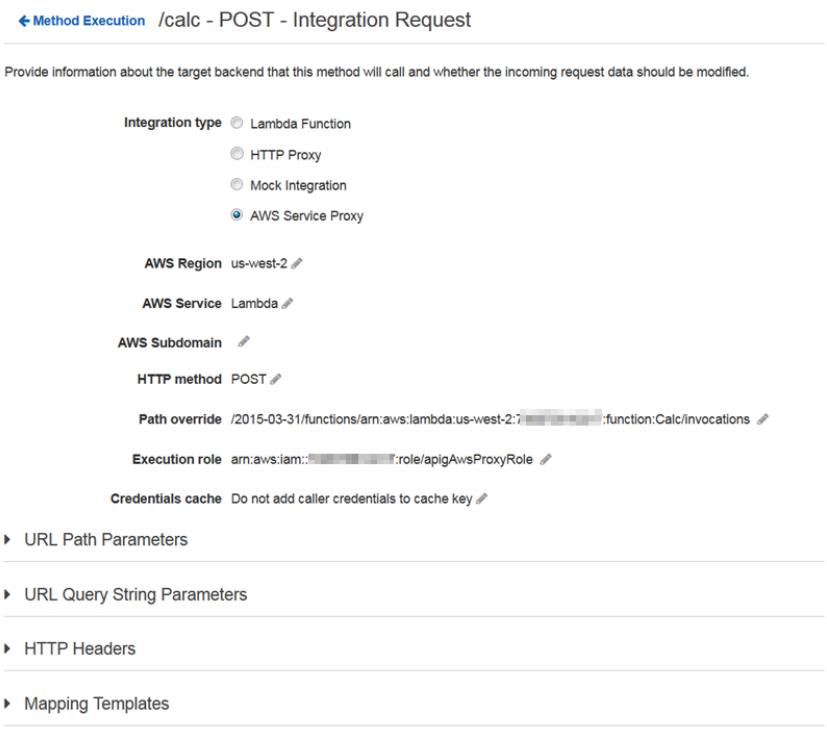
Create a POST Method with a JSON Payload to Call the Lambda Function

The following steps describe how to expose a POST method with a JSON payload.

To set up the POST method with a JSON payload to invoke a Lambda function

1. Choose **Create Method** in the API Gateway console to create a POST method for the **LambdaGate** API's **/calc** resource.

   In the method's **Set Up** panel, configure the POST method with the following settings.



Using a POST request with a JSON payload is the simplest way to invoke a Lambda function, because no mappings are needed.

2. 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 POST as an asynchronous call, you can add an `InvocationType:Event` header in the method request and map it to the `X-Amz-Invocation-Type` header in the integration request, using the header mapping expression of `method.request.header.InvocationType`. You must inform the clients to include the `InvocationType:Event` header in the method request. Alternatively, you can set the `X-Amz-Invocation-Type` header with the `'Event'` string literal in the integration

request, without requiring the client to include the header. The asynchronous call will return an empty response, instead.

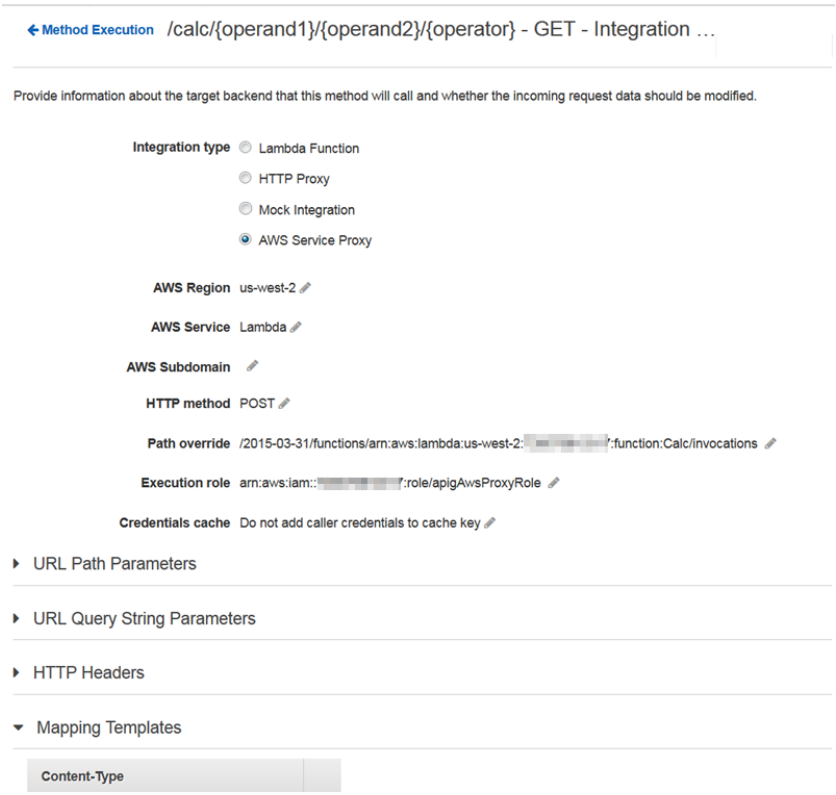
Create a GET Method with Path Parameters to Call the Lambda Function

The following steps describe how to set up the GET method with path parameters to call the Lambda function.

To set up the GET method with URL path parameters to invoke the Lambda function

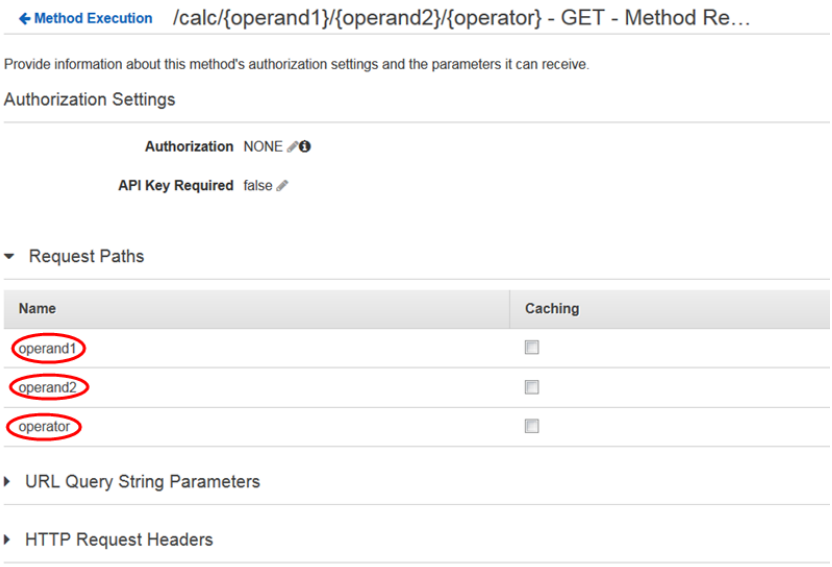
1. Choose **Create Method** in the API Gateway console to create a GET method for the API's **/calc/{operand1}/{operand2}/{operator}** resource.

   In the method's **Set up** pane, configure this GET method with the following settings.

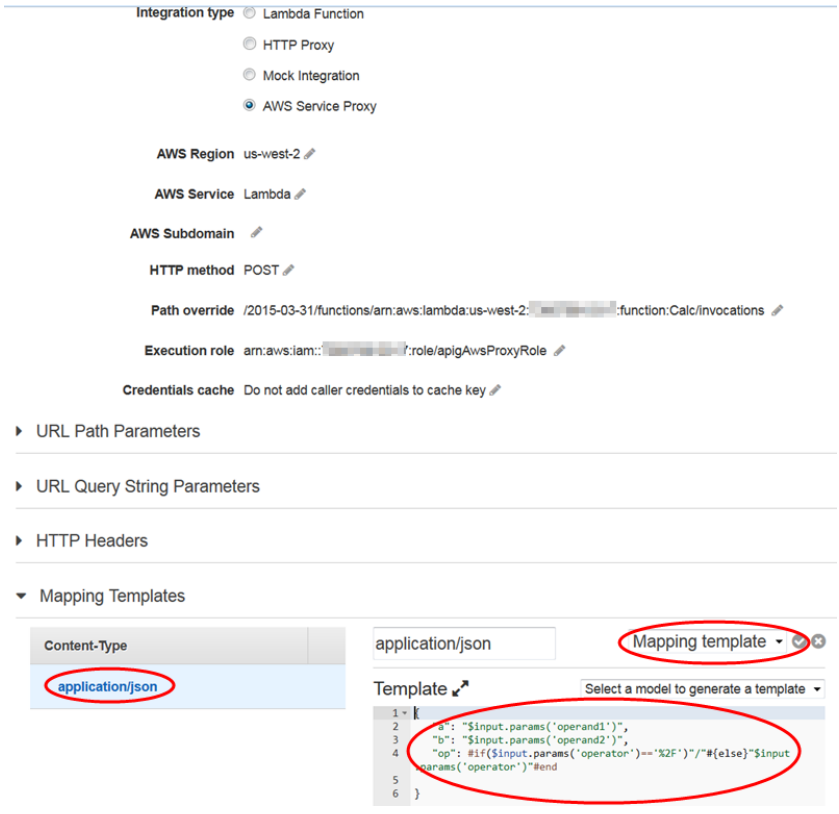


   Next, we will set up **Mapping Templates** to translate the URL path parameters into the integration request JSON payload as the input to the Lambda function.

2. In **Method Request** for the **GET** method on **/calc/{operand1}/{operand2}/{operator}**, expand the **Request Paths** section to verify that the path parameters are there.



3. Go back to **Integration Request**. Expand the **Mapping Templates** section. If necessary, in **Content-Type**, under **application/json**, choose **Add mapping template**.



Type the following in the **Mapping Template** editor:

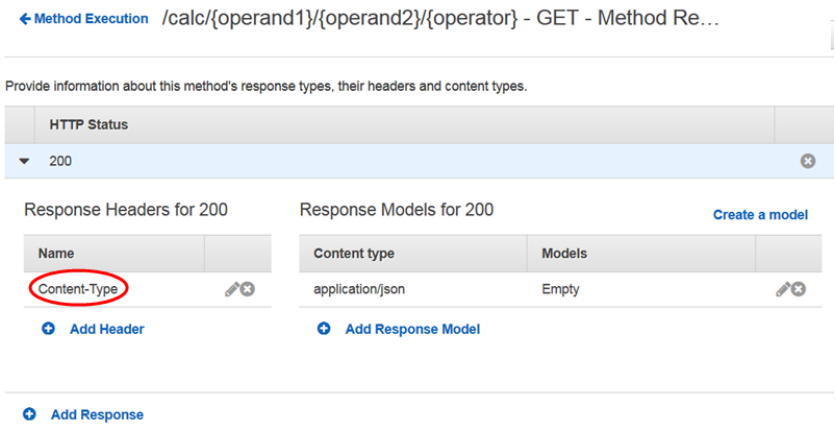
```
{
```

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

}
```

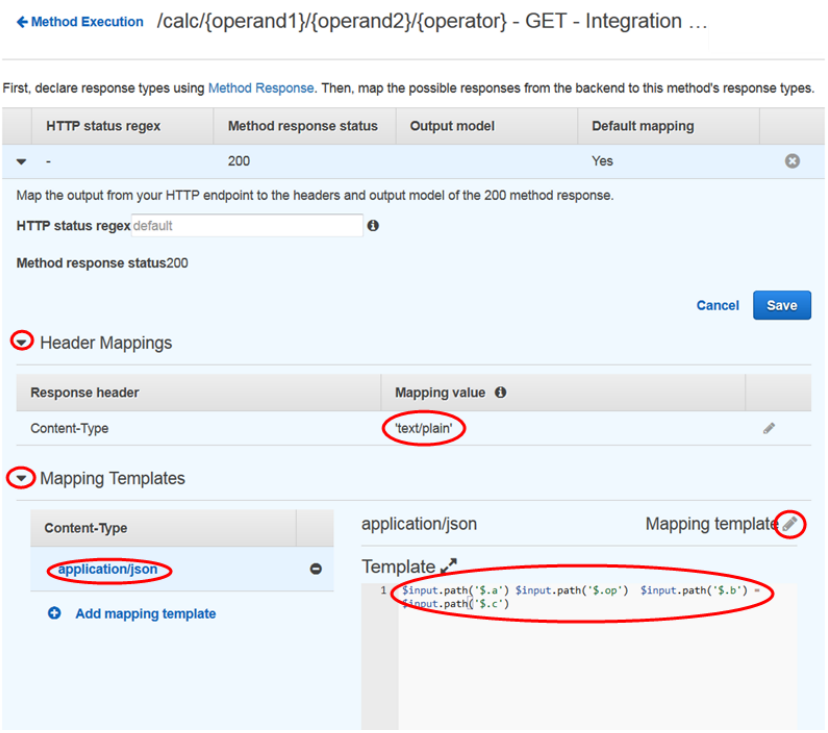
This template maps the three URL path parameters, declared when the **/calc/{operand1}/{operand2}/{operator}** resource was created, into designated property values of the JSON object. Because URL paths must be URL-encoded, the division operator must be specified as `%2F` instead of `/`. This template maps these translations as well. The transformed JSON object will be included as the integration request payload.

4. As another exercise, we demonstrate how to translate the JSON returned from the Lambda function to show the output as a plain text string to the caller. This involves resetting the method request's Content-Type header to "text/plain" and providing a mapping template to translate the JSON output into a plain string.

   First, make sure that **Content-Type** header is included in the **Response Headers for 200** section in **Method Response**.



5. In **Integration Response**, expand the 200 method response entry. Expand the **Header Mappings** section. In **Mapping value** for **Content-Type**, type **`'text/plain'`**. This header mapping expression overrides the Content-Type header with a literal string, which must be enclosed within a pair of single quotes.



Next, expand the **Mapping Templates** section, highlight the **application/json** entry under the **Content-Type** header (of integration response), open the **Mapping template** editor, enter and save the following mapping script:

```
$input.path('$.a') $input.path('$.op') $input.path('$.b') = $input.path('$.c')
```

6. Choose **Test** to verify the GET method on the **/calc/{operand1}/{operand2}/{operator}** works as expected. The following request URL:

```
/calc/1/2/%2F
```

should produce the following plain text output:

```
1 / 2 = 0.5
```

Note
As part of a URL, the division operator (/) is URL-encoded (%2F).

7. After testing the API using the Test Invoke in the API Gateway console, you must deploy the API to make it public available. If you update the API, such as adding, modifying or deleting a resource or method, updating any data mapping, you must redeploy the API to make the new features or updates available.

A Sample API as a Lambda Proxy in Swagger with API Gateway Extensions

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-02-23T05:35:54Z",
    "title": "LambdaGate"
  },
  "host": "a123456789.execute-api.us-east-1.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/calc": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "operand2",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "operator",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "operand1",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "operand_1": {
                "type": "string"
              },
              "operand_2": {
                "type": "string"
              },
              "operator": {
                "type": "string"
```

```
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.operator": "integration.re
sponse.body.op",
                "method.response.header.operand_2": "integration.re
sponse.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 => $res.c\"\n}"
              }
            }
          },
          "requestTemplates": {
           "application/json": "{\n    \"a\":  \"$input.params('operand1')\",\n
    \"b\":  \"$input.params('operand2')\", \n    \"op\": \"$input.params('oper
ator')\"   \n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path//2015-03-31/func
tions/arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "httpMethod": "POST",
          "type": "aws"
        }
      },
      "post": {
        "produces": [
          "application/json"
        ],
        "parameters": [],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {}
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "__passthrough__"
              }
            }
          },
```

```
          "uri": "arn:aws:apigateway:us-west-2:lambda:path//2015-03-31/func
tions/arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "httpMethod": "POST",
          "type": "aws"
        }
      }
    },
    "/calc/{operand1}/{operand2}/{operator}": {
      "get": {
        "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": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type": "'text/plain'"
              },
              "responseTemplates": {
                "application/json": "\"$input.path('$.a') $input.path('$.op')
 $input.path('$.b') = $input.path('$.c')\""
              }
            }
```

```
          },
          "requestTemplates": {
           "application/json": "\n{\n   \"a\": \"$input.params('operand1')\",\n
   \"b\": \"$input.params('operand2')\",\n   \"op\": #if($input.params('operat
or')=='%2F')\"/\"#{else}\"$input.params('operator')\"#end\n   \n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path//2015-03-31/func
tions/arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "httpMethod": "POST",
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object"
    }
  }
}
```

Create an API Gateway API as an Amazon Kinesis Proxy

This section describes how to create and configure an API Gateway API as an AWS proxy to access Amazon Kinesis.

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

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

1. List the user's available streams in Amazon 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 Amazon Kinesis
2. The `CreateStream`, `DescribeStream`, or `DeleteStream` action
3. The `GetRecords` or `PutRecords` (including `PutRecord`) action in Amazon Kinesis

Specifically, we will 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 Amazon 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 Amazon 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 Amazon 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 Amazon 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 Amazon 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 Amazon 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 Amazon 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 Amazon Kinesis. This helper method must be supplied to the `ListStreams` action in Amazon Kinesis.

You can apply the instructions presented here to other Amazon Kinesis actions. For the complete list of the Amazon 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 (p. 110). 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 http://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 Amazon Kinesis

To allow the API to invoke Amazon 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 Amazon Kinesis, you can use the AmazonKinesisReadOnlyAccess policy that allows the `Get*`, `List*`, and `Describe*` actions in Amazon Kinesis to be invoked.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "kinesis:Get*",
                "kinesis:List*",
                "kinesis:Describe*"
            ],
            "Resource": "*"
        }
    ]
}
```

This policy is available from the IAM console and its ARN is `arn:aws:iam::aws:policy/AmazonKinesisReadOnlyAccess`.

To enable read-write actions in Amazon Kinesis, you can use the AmazonKinesisFullAccess policy.

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

This policy is also available from the IAM console. 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 an Amazon 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 Amazon Kinesis

1. In the API Gateway console, choose **Create API**.
2. In **API name**, type **`KinesisProxy`**. Leave the default values in the other fields.
3. For **Clone from API**, choose **Do not clone from existing API**.
4. Type a description in **Description**.
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 Amazon Kinesis

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

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

To list Amazon Kinesis streams by using the API Gateway console

1. Select the API root resource. In **Actions**, choose **Create Resource**.

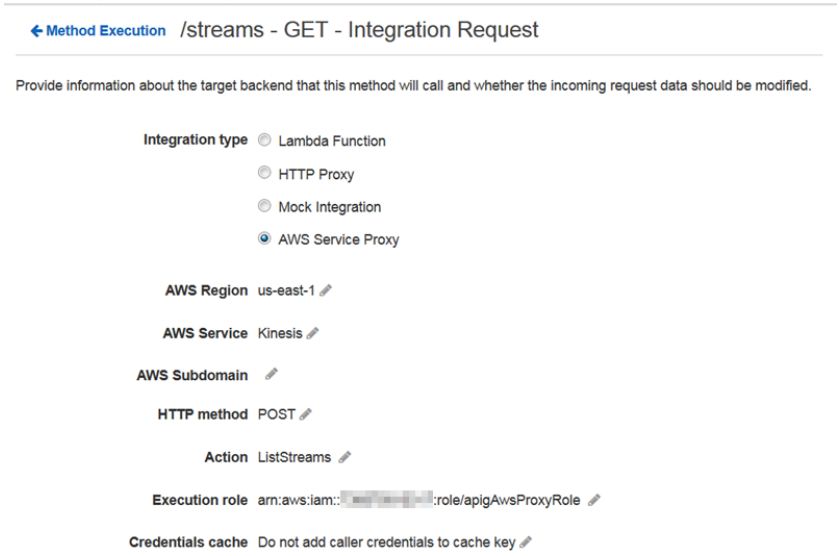
   In **Resource Name**, type `Streams`, leave **Resource Path** as the default, and choose **Create Resource**.
2. Select the `/Streams` resource. From **Actions**, choose **Create Method**, choose **GET** from the list, and then choose the checkmark icon to finish creating the method.

   Note
   You can choose any of the available HTTP verbs for a method request. We use `GET` here, because listing streams is a READ operation.
3. In the method's **Setup** pane, choose **Show Advanced** and then choose **AWS Service Proxy**.

   a. For **AWS Region**, choose a region (e.g., **us-east-1**).
   b. For **AWS Service**, choose **Kinesis**.
   c. For **HTTP method**, choose **POST**.

      Note
      For the integration request with Amazon Kinesis, you must choose the `POST` HTTP verb to invoke the action, although you can use any of the available HTTP verbs for the API's method request.

   d. For **Action Type**, choose **Use action name**.
   e. For **Action**, type **`ListStreams`**.
   f. For **Execution role**, type the ARN for your execution role.
   g. Choose **Save** to finish the initial setup of the method.



The initial setup of the integration request will suffice if there is no need to map data between the method and integration requests and/or between the method and integration responses. Examples discussed in this topic require data mapping, which is covered in the second half of the **Integration Request** pane.

4. 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 checkmark icon to save the setting.

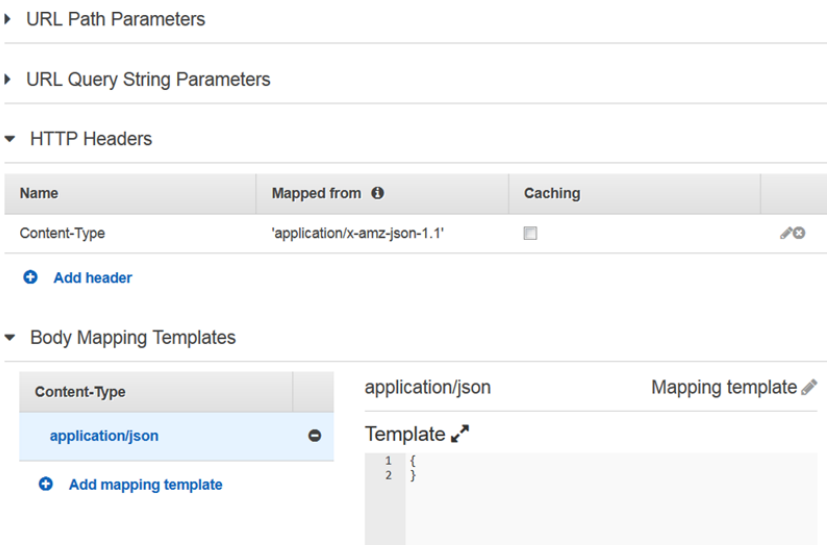
5. Expand the **Body Mapping Templates** section:

   a. Choose **Add mapping template**.
   b. For **Content-Type**, type `application/json`.
   c. Choose the checkmark icon to save the setting.
   d. Choose the pencil icon to the right of **Mapping template**.
   e. Choose **Mapping template** from the drop-down list to open the **Template** editor.
   f. Type **{}** in the template editor.
   g. Choose the checkmark icon 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.



6. Test the GET method on the Streams resource to invoke the `ListStreams` action in Amazon 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 have already created two streams named "myStream" and "yourStream" in Amazon Kinesis, the successful test will return a 200 OK response containing the following payload:

```
{
     "HasMoreStreams": false,
     "StreamNames": [
         "myStream",
         "yourStream"
     ]
}
```

Create, Describe, and Delete a Stream in Amazon Kinesis

Creating, describing, and deleting a stream in Amazon Kinesis involves making the following Amazon 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"
}
```

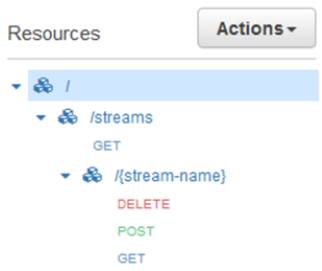
```
POST /?Action=DescribeStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "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 our 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 will create our API slightly differently.

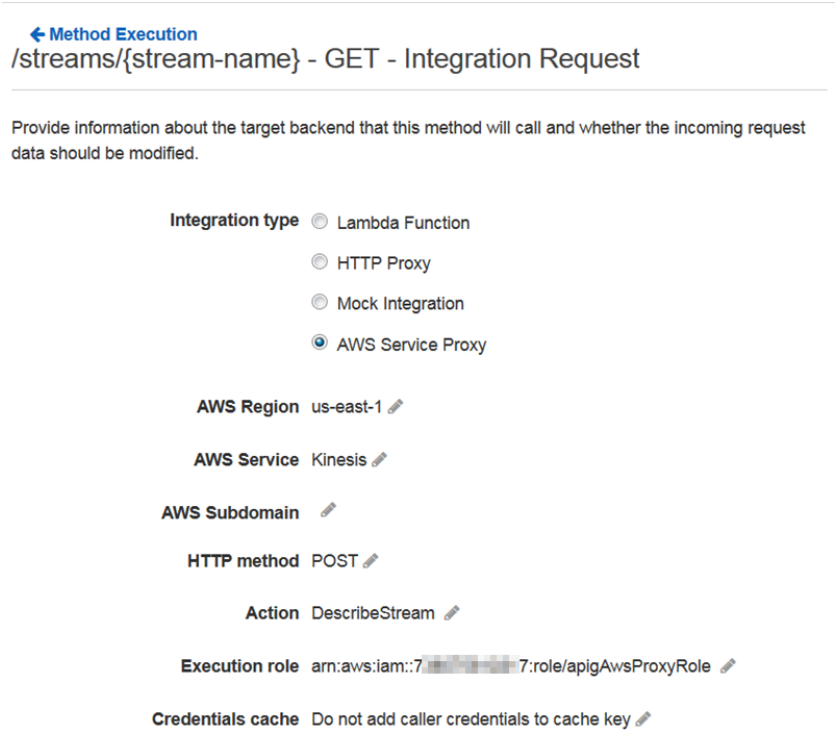
We will expose the `GET`, `POST`, and `Delete` HTTP methods on a to-be-named `Stream` resource. We will use the `{stream-name}` path variable to hold the to-be-named stream resource and integrate these API methods with the Amazon Kinesis' `DescribeStream`, `CreateStream`, and `DeleteStream` actions, respectively. We require that the client pass other input data as headers, query parameters, or the payload of a method request, and we provide mapping templates to transform the data to the required integration request payload.

After the methods are created on a to-be-named stream resource, the structure of the API looks like the following:

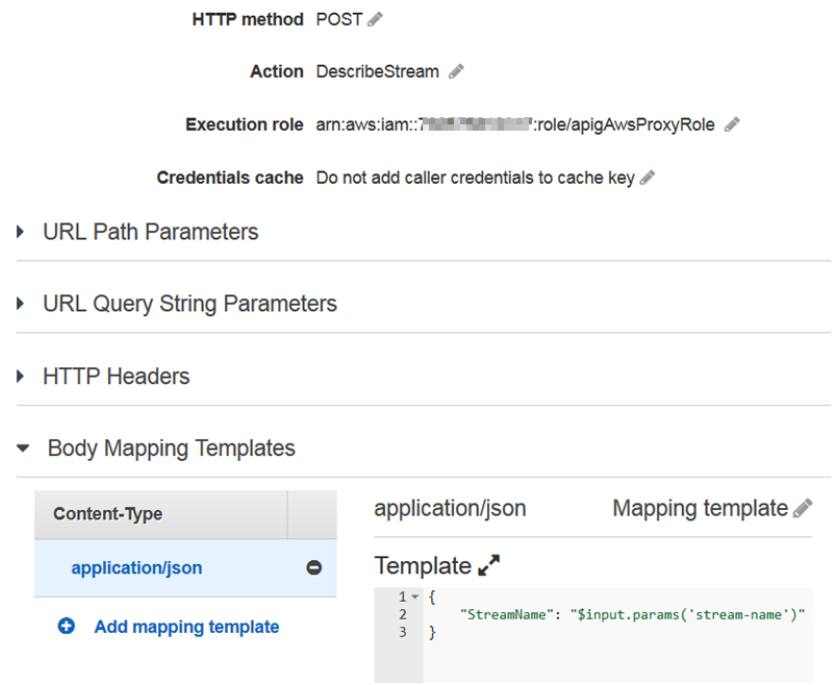


To configure and test the GET method on a stream resource

1. Set up the GET method to describe a named stream in Amazon Kinesis, as shown in the following.



2. Map data from the GET method request to the integration request, as shown in the following:



3. Test the GET method to invoke the `DescribeStream` action in Amazon 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 Amazon 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:

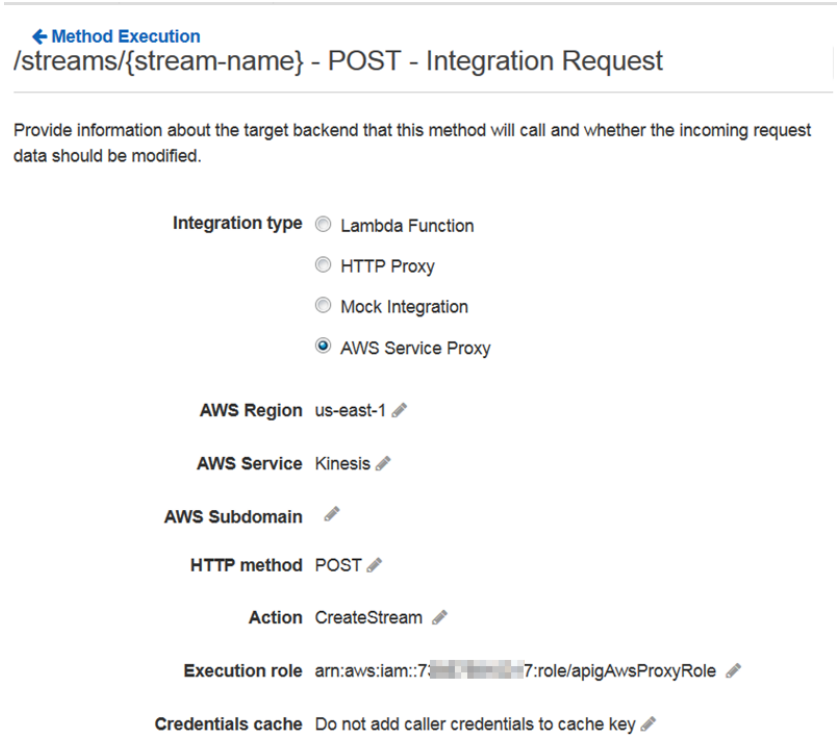
```
{
  "StreamDescription": {
    "HasMoreShards": false,
    "RetentionPeriodHours": 24,
    "Shards": [
      {
        "HashKeyRange": {
          "EndingHashKey": "680564733841876926926749214863536422 90",
          "StartingHashKey": "0"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
"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:

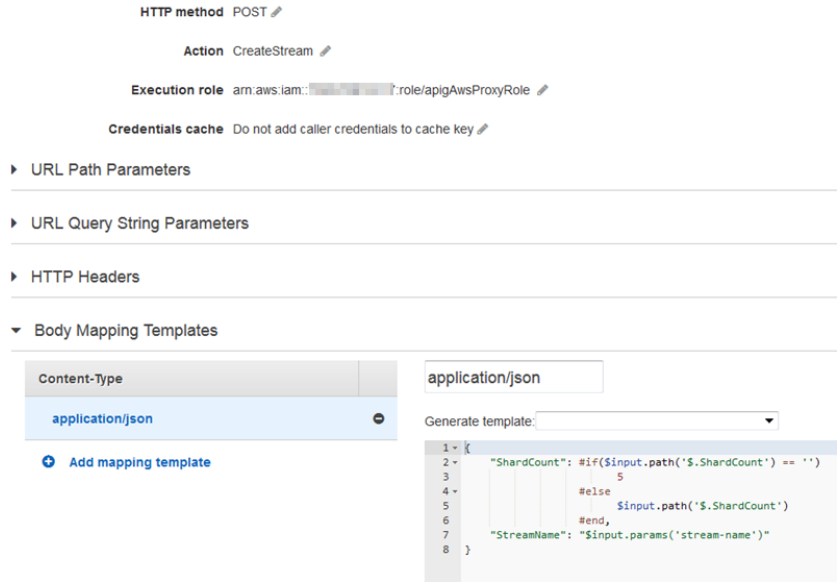
```
GET https://your-api-id.execute-api.region.amazonaws.com/stage/streams/myS
tream HTTP/1.1
Host: your-api-id.execute-api.region.amazonaws.com
Content-Type: application/json
Authorization: ...
X-Amz-Date: 20160323T194451Z
```

To configure and test the POST method on a stream resource

1. Set up the POST method on a stream resource to create the stream in Amazon Kinesis, as shown in the following:



2. Map data from the POST method request to the integration request, as shown in the following:



In this example, we use the following mapping template to set `ShardCount` to a fixed value of 5 if the client does not specify a value in the method request payload. Otherwise, we pass the client-supplied value to the back end.

```
{
    "ShardCount": #if($input.path('$.ShardCount') == '') 5 #else $in
put.path('$.ShardCount')",
    "StreamName": "$input.params('stream-name')"
}
```

The preceding `if ( $input.path('$.ShardCount') == '' ) ...` Boolean expression tests if the method request's JSON payload does not have the `ShardCount` property declared or if the property value is empty.

3. Test the POST method to create a named stream in Amazon Kinesis:

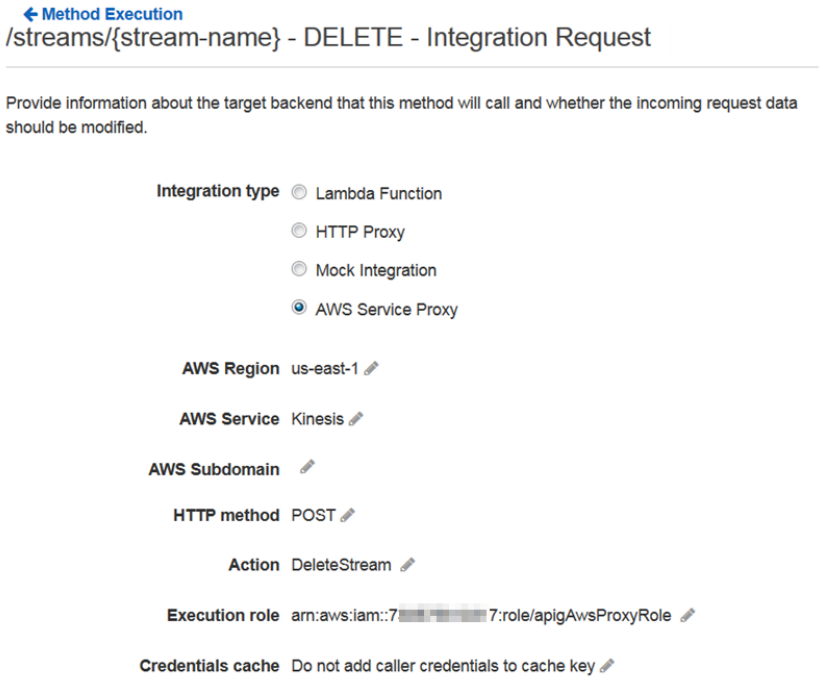
   From the API Gateway console, select **/streams/{stream-name}/POST** in the **Resources** pane, choose **Test** to start testing, type the name of an existing Amazon Kinesis stream in **Path** for `stream-name`, and choose **Test**. If the test is successful, a 200 OK response is returned with no data.

   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 Amazon Kinesis:

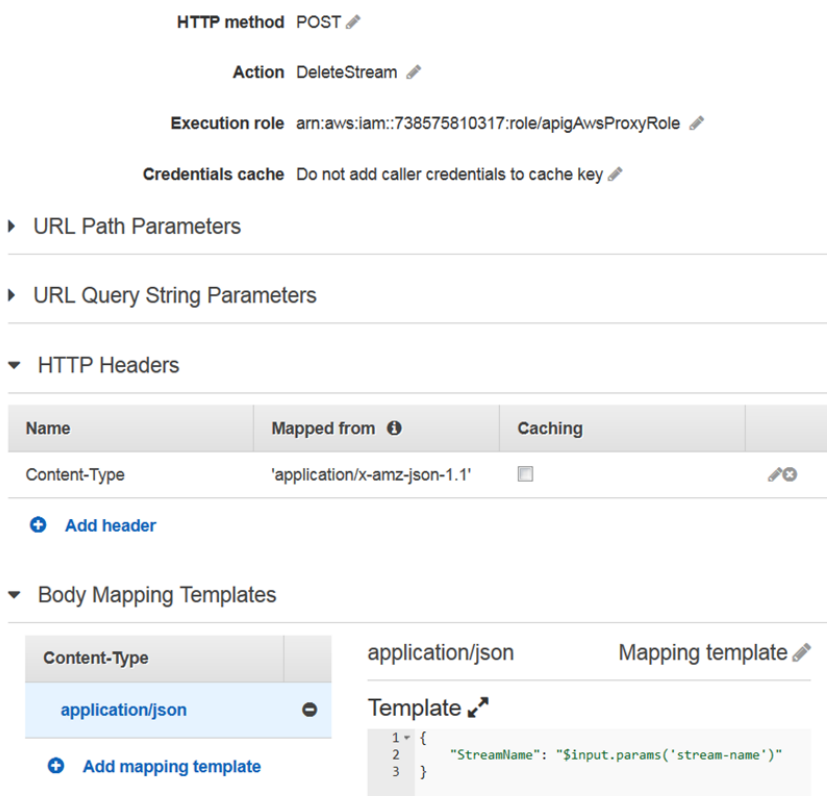
```
POST https://your-api-id.execute-api.region.amazon
aws.com/stage/streams/yourStream HTTP/1.1
Host: your-api-id.execute-api.region.amazonaws.com
Content-Type: application/json
Authorization: ...
X-Amz-Date: 20160323T194451Z

{
    "ShardCount": 5
}
```

Configure and test the DELETE method on a stream resource

1. Set up the DELETE method to invoke the `DeleteStream` action in Amazon Kinesis, as shown in the following.



2. Map data from the DELETE method request to the integration request, as shown in the following:



3. Test the DELETE method to delete a named stream in Amazon 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 Amazon 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 Amazon Kinesis:

```
DELETE https://your-api-id.execute-api.region.amazon
aws.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 Amazon Kinesis

After you create a stream in Amazon 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 Amazon 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 Amazon 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
}
```

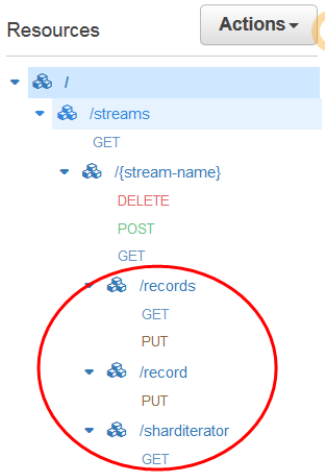
Here, the source stream from which we are getting records is specified in the required `ShardIterator` value, as is shown in the following Amazon Kinesis action to obtain a shard iterator:

```
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`).

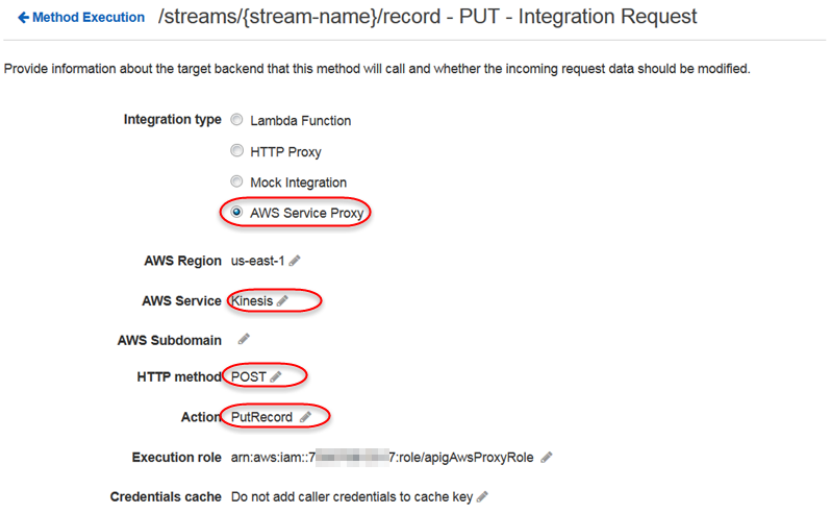
The following figure shows the API structure of resources after the methods are created:



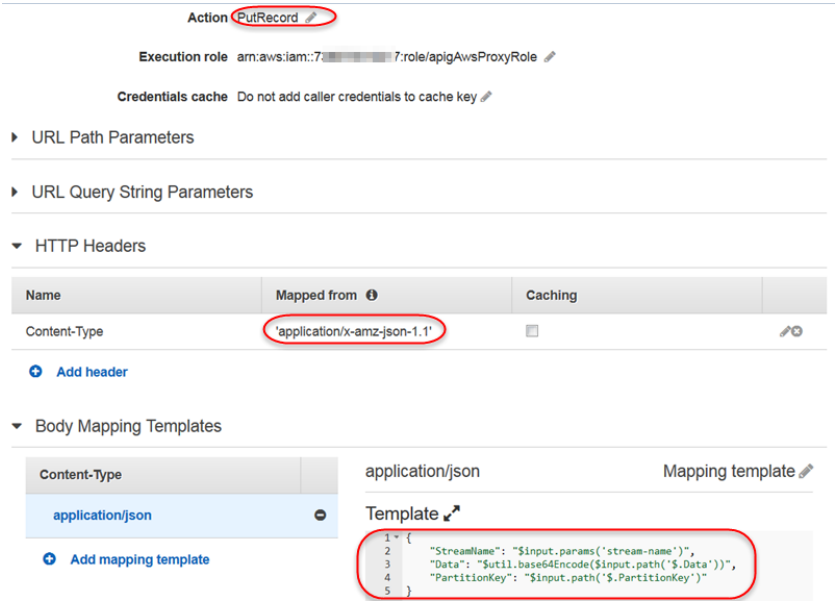
The following four procedures describe how to set up each of the methods, how to map data from the method requests to the integration requests, and how to test the methods.

To configure and test the PUT method on the record resource in the API to invoke the PutRecord action in Amazon Kinesis:

1. Set up the PUT method, as shown in the following:



2. Configure data mapping for the PUT-on-Record method, as shown in the following:



The preceding 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.

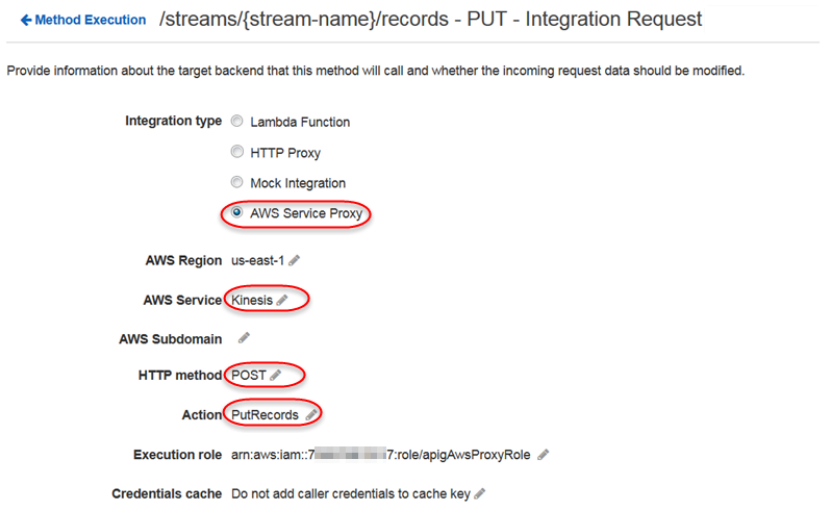
3. To test the PUT method, set the `stream-name` path variable to an existing stream, supply a payload of the preceding format, and then submit the method request. The successful result is a 200 OK response with a payload of the following format:

```
{
  "SequenceNumber":
"495594099445378808501333454601698865935731021151679283 86",
  "ShardId": "shardId-000000000004"
```

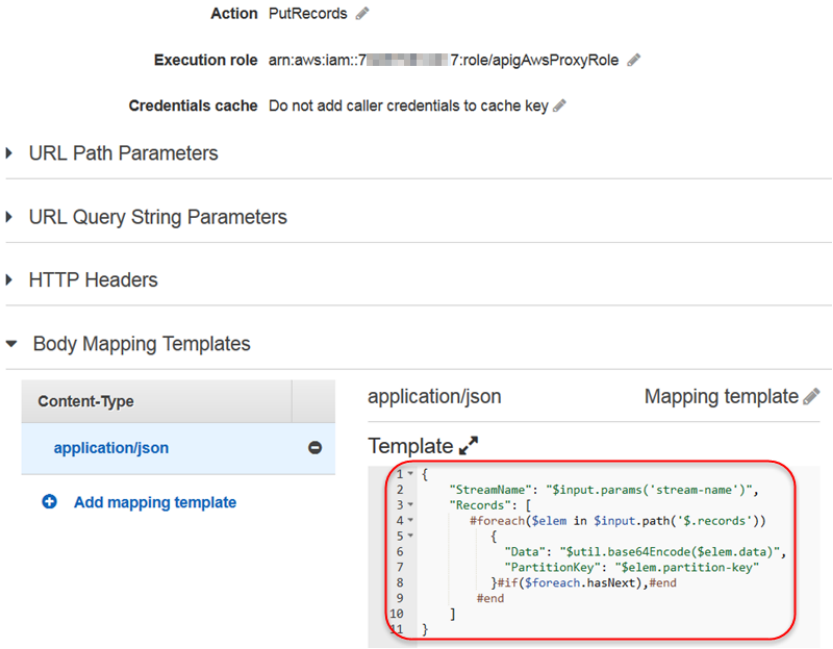
```
}
```

To configure and test the PUT method on the records resource in the API to invoke the PutRecords action in Amazon Kinesis

1. Set up the PUT method, as shown in the following:



2. Configure data mapping for the PUT method, as shown in the following:



The preceding mapping template assumes the method request payload can be modeled by 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" }
        }
      }
    }
  }
}
```

3. To test the PUT method, set the `stream-name` path variable to an existing stream, supply a payload as previously shown, and submit the method request. The successful result is a 200 OK response with a payload of the following format:

```
{
    "records": [
        {
            "data": "some data",
            "partition-key": "some key"
        },
        {
            "data": "some other data",
            "partition-key": "some key"
        }
    ]
}
```

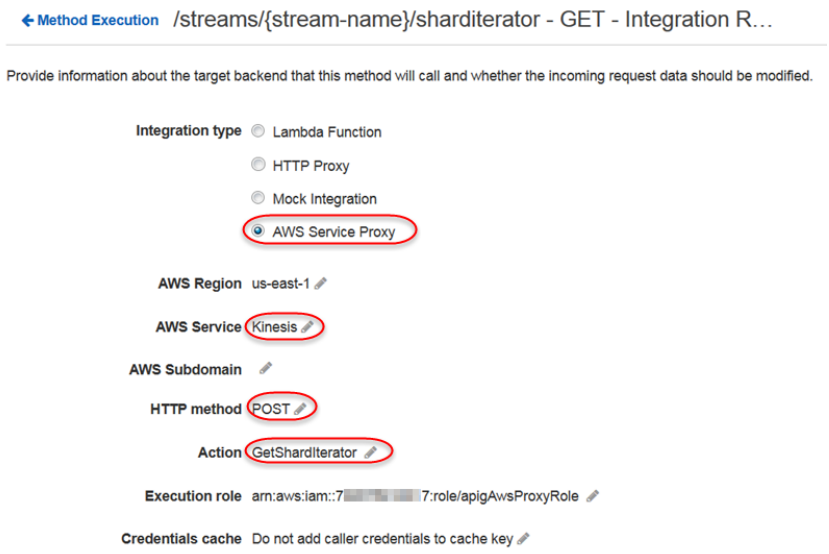
The response payload will be similar to the following output:

```
{
  "FailedRecordCount": 0,
  "Records": [
    {
      "SequenceNumber":
"49559409944537880850133345460167468741933742152373764162",
      "ShardId": "shardId-000000000004"
    },
    {
      "SequenceNumber":
"49559409944537880850133345460168677667753356781548470338",
      "ShardId": "shardId-000000000004"
    }
  ]
}
```

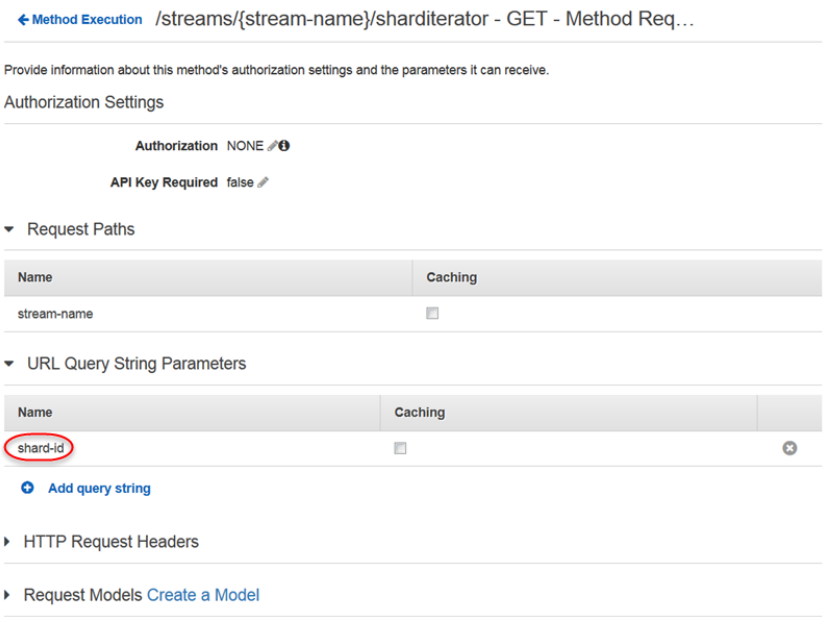
To configure and test the GET method on the ShardIterator resource in the API to invoke the GetShardIterator action in Amazon Kinesis

The GET-on-ShardIterator method is a helper method to acquire a required shard iterator before calling the GET-on-Records method.

1. Set up the GET-on-ShardIterator method, as shown in the following:

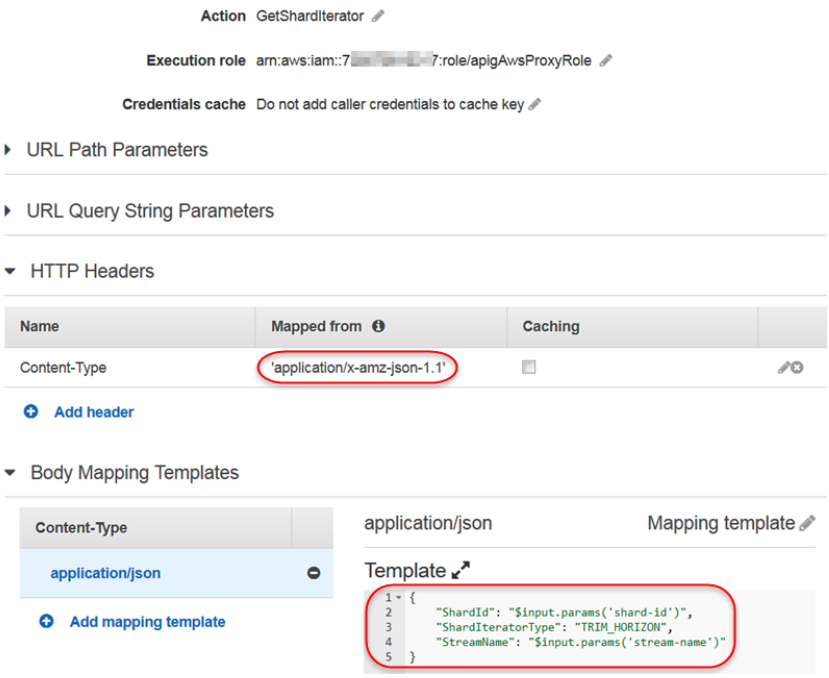


2. The `GetShardIterator` action requires an input of a ShardId value. To pass a client-supplied `ShardId` value, we add a `shard-id` query parameter to the method request, as shown in the following:



In the following mapping template, we add the translation of the `shard-id` query parameter value to the `ShardId` property value of the JSON payload for the `GetShardIterator` action in Amazon Kinesis.

3. Configure data mapping for the GET-on-ShardIterator method:



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**.

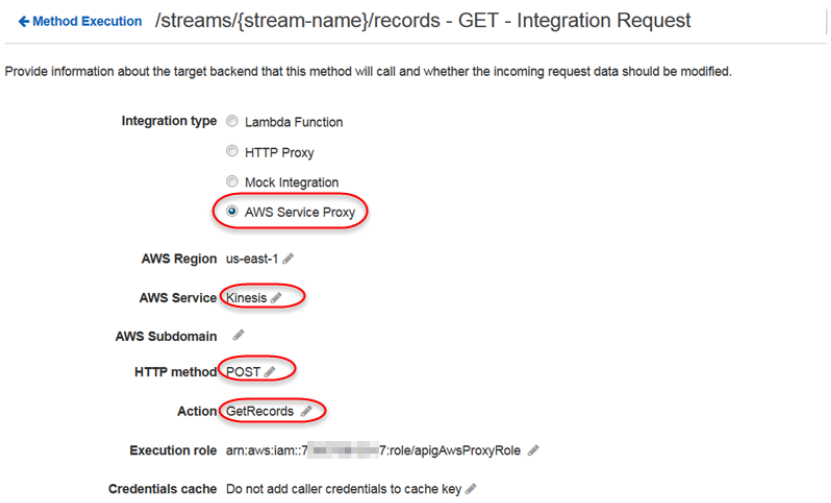
   The successful response payload will be similar to the following output:

```
{
  "ShardIterator": "AAAAAAAAAAFYVN3VlFy..."
}
```

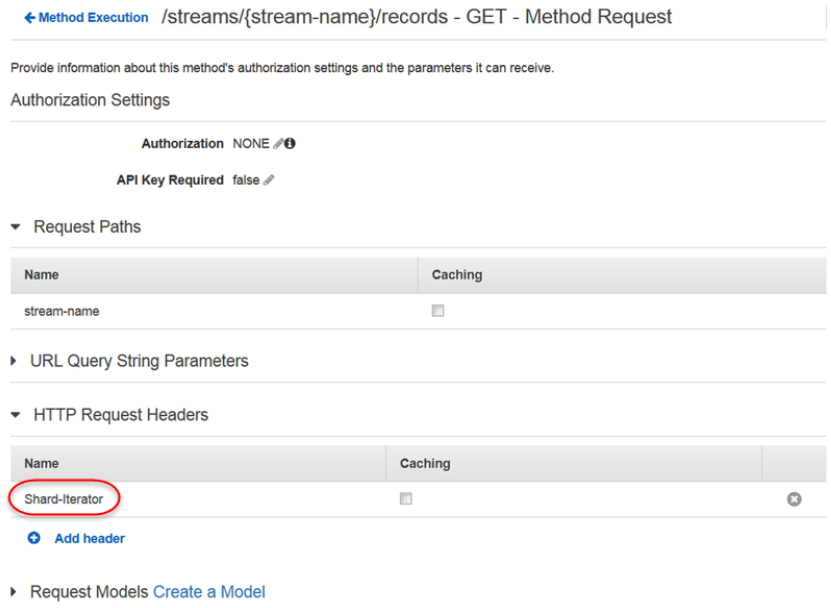
   Make note of the `ShardIterator` value. You will need it to get records from a stream.

To configure and test the GET Method on the records resource in the API to invoke the GetRecords action in Amazon Kinesis

1. Set up the GET method, as shown in the following:

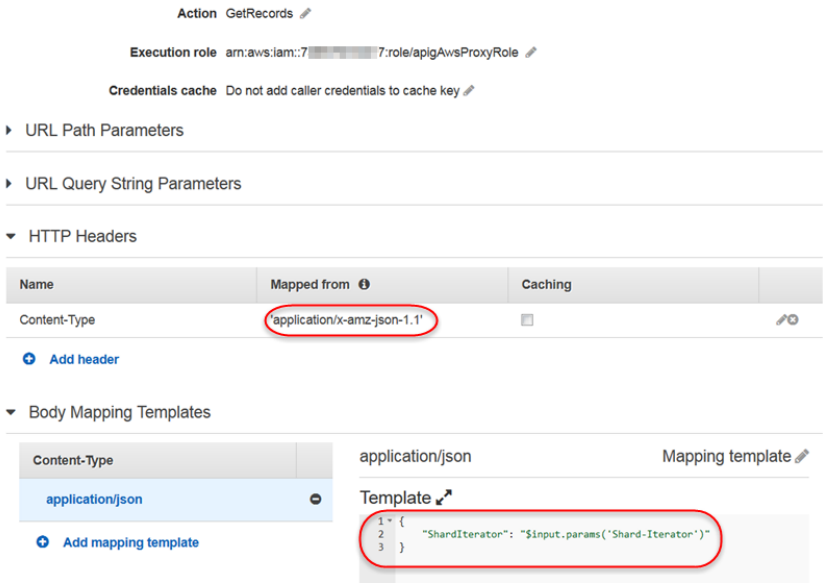


2. The `GetRecords` action requires an input of a `ShardIterator` value. To pass a client-supplied `ShardIterator` value, we add a `Shard-Iterator` header parameter to the method request, as shown in the following:



In the following mapping template, we add the mapping from the `Shard-Iterator` header value to the `ShardIterator` property value of the JSON payload for the `GetRecords` action in Amazon Kinesis.

3. Configure data mapping for the GET-on-Records method:



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-on-ShardIterator method (above), and choose **Test**.

   The successful response payload will be similar to the following output:

```
{
  "MillisBehindLatest": 0,
  "NextShardIterator": "AAAAAAAAAAF...",
  "Records": [ ... ]
}
```

Swagger Definitions of a Sample API as an Amazon Kinesis Proxy

```
{
  "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": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n}"
          },
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
          "httpMethod": "POST",
          "requestParameters": {
           "integration.request.header.Content-Type": "'application/x-amz-json-
1.1'"
          },
          "type": "aws"
        }
      }
    },
    "/streams/{stream-name}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$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}"
          },
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream",

          "httpMethod": "POST",
          "type": "aws"
        }
      },
      "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}"
          },
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/CreateStream",
          "httpMethod": "POST",
          "requestParameters": {
           "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.head
er.Content-Type"
              }
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
              "method.response.header.Content-Type": "integration.response.head
```

```
er.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500",
              "responseParameters": {
              "method.response.header.Content-Type": "integration.response.head
er.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    \"Data\": \"$util.base64Encode($input.path('$.Data'))\",\n
\"PartitionKey\": \"$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\": \"$in
put.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": "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": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Records\": [\n       #foreach($elem in $input.path('$.re
cords'))\n          {\n            \"Data\": \"$util.base64En
code($elem.data)\",\n            \"PartitionKey\": \"$elem.partition-key\"\n
        }#if($foreach.hasNext),#end\n        #end\n    ]\n}",
            "application/x-amz-json-1.1": "#set($inputRoot = $in
put.path('$'))\n{\n  \"StreamName\": \"$input.params('stream-name')\",\n
```

```
\"records\" : [\n    #foreach($elem in $inputRoot.records)\n    {\n
\"Data\" : \"$elem.data\",\n       \"partition-key\" : \"$elem.partition-key\"\n
    }#if($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"
    }
  }
}
```

Controlling Access 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. Custom authorizers and Amazon Cognito user pools provide customizable authorization and authentication solutions.

Topics

Set IAM Permissions to Access API Gateway

Topics

Control Access to API Gateway with IAM Policies

When working with Amazon API Gateway, you access two services. You use one to create, configure, deploy and update your API and the other to actually execute your deployed API upon requests by a client. When setting access permissions in an IAM policy, you reference the API managing service as `apigateway` and the API executing service as `execute-api`. The `apigateway` service supports the actions of `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `OPTIONS`, `HEAD` and the `execute-api` service supports the `Invoke` and `InvalidateCache` actions. To create an IAM policy using the Policy Generator in the IAM console, select **Manage Amazon API Gateway** as **AWS Service** to set permissions statements for

`apigateway` and select **Amazon API Gateway** as **AWS Service** to set permission statements for `execute-api`.

You can use IAM to allow IAM users and roles in your AWS account to manage only certain API Gateway entities (for example, APIs, resources, methods, models, and stages) and perform only certain actions against those entities. You may want to do this, for example, if you have IAM users you want to allow to list, but not create, resources and methods for selected APIs. You may have other IAM users you want to allow to list and create new resources and methods for any API they have access to in API Gateway.

In the Get Ready to Use API Gateway (p. 4) instructions, you attached an access policy to an IAM user in your AWS account that contains a policy statement similar to this:

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

This statement allows the IAM user in your AWS account to perform all available actions and access all available resources in API Gateway to which your AWS account has access. In practice, you may not want to give the IAM users in your AWS account this much access.

You can also use IAM to enable users inside your organization to interact with only certain API methods in API Gateway.

In the Configure How a User Calls an API Method (p. 65) instructions, the API Gateway console may have displayed a resource ARN you used to create a policy statement similar to this:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:my-aws-account-id:my-api-id/my-
stage/GET/my-resource-path"
      ]
    }
  ]
}
```

This statement allows the IAM user to call the GET method for the resource path associated with the specified resource ARN in API Gateway. In practice, you may want to give IAM users access to more methods.

Note
IAM policies are effective only if IAM authentication is enabled. If you, as the API owner, has enabled AWS identity and access management on a specific resource, users from other AWS accounts cannot access your API. If you do not enable IAM authentication on the resource, that resource is effectively public accessible.

Create and Attach a Policy to an IAM User

To create and attach an access policy to an IAM user that restricts the API Gateway entities the IAM user can manage or the API methods the IAM user can call, do the following:

1. Sign in to the Identity and Access Management (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 will be easy for you to refer to later, if needed.
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 additional statements as needed, 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 will not have permissions unless a corresponding `Allow` statement is explicitly stated.)
6. Choose **Users**.
7. Choose the IAM user to whom you want to attach the policy.
8. For **Permissions**, for **Managed Policies**, choose **Attach Policy**.
9. Select the policy you just created, and then choose **Attach Policy**.

Statement Reference of IAM Policies for Managing API in API Gateway

The following information describes the `Action` and `Resource` format used in an IAM policy statement to grant or revoke permissions for managing API Gateway API entities, such as restapis, resources, methods, models, stages, custom domain names, API keys, etc.

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*.

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 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 Access to API Gateway with IAM Policies (p. 188).

Topics

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:*::/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/*/stages/*"
            ]
        }
    ]
}
```

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 /respais` to query the available APIs. Also, if `arn:aws:apigateway:us-east-1::/restapis/*/stages` 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 to API Gateway with IAM Policies (p. 188).

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/test/*/petstorewalkthrough/pets"
      ]
    }
  ]
}
```

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:'*'` header in at least its 200 response.

> **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 API.

Topics

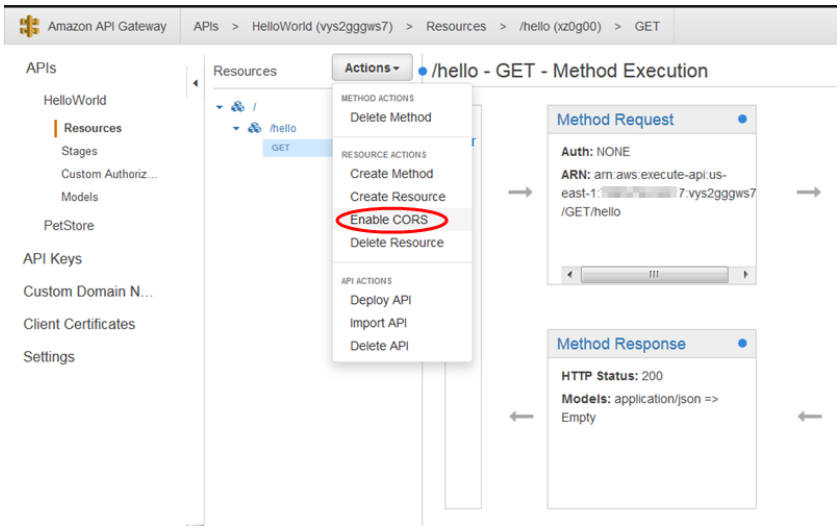
Prerequisites

- You must have the method available in API Gateway. For instructions on how to create and configure a method, see Build an API Gateway API Step by Step (p. 14).

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.

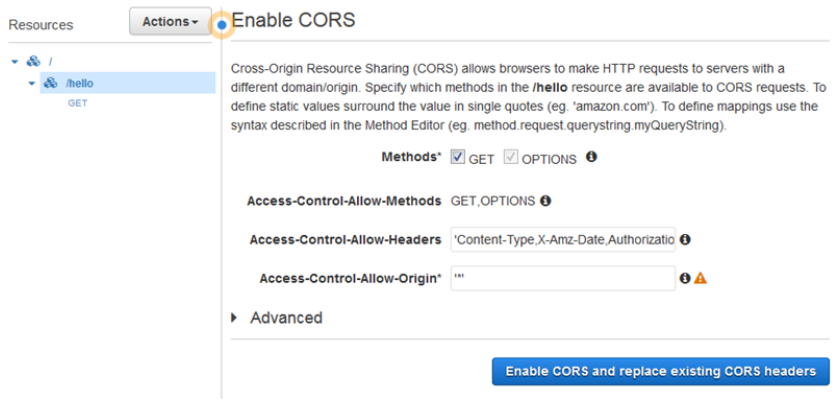


5. In the **Enable CORS** form, do the following:

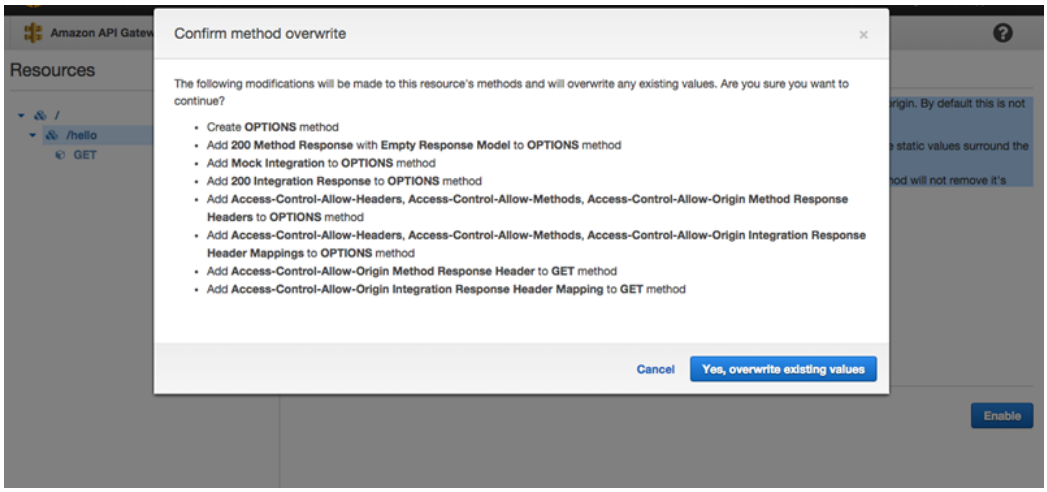
   a. In the **Access-Control-Allow-Headers** input field, type a static string of a comma-separated list of headers that the client must submit in the actual request of the resource. Use the console-provided header list of `'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'` or specify your own headers.

   b. 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.

   c. Choose **Enable CORS and replace existing CORS headers**.



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.

Enable CORS for a Resource Using the API Gateway Import API

If you are using the API Gateway Import API (p. 110), 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 Configure Mock Integration for a Method (p. 69).

```
/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 an API Key in API Gateway

You can use an API key in API Gateway to identify apps calling the API and control API access based on the API key. You can use an API key to control how an API is used. For example, you can generate an API key and give it to specific app developers to make the API available for their app users. When an API key is enabled, API calls must contain the specified key, as the value of the `x-api-key` header of the requests. Requests without the matching API key will then be rejected. API keys are useful to control that an API is used as expected and curtail abusive uses by changing the API keys. They should not be treated as a security mechanism for controlling access to an API.

Topics

Prerequisites

1. You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).
2. You must have deployed the API in API Gateway at least once. Follow the instructions in Deploying an API (p. 221).

Use an API Key with the API Gateway Console

To enable an API key with the API Gateway console, follow these instructions:

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose the **GET** method under a resource of your choosing.
3. Choose the **Method Request** box
4. If **API Key Required** is set to **false**, choose the pencil icon next to it. From the drop-down menu list, choose **true**. Finally, choose the check-mark icon to save the setting.

   Note
   The steps above configures the API Gateway to enforce using API key. Otherwise, the API key created following the instructions below will not be used.
5. In the secondary navigation bar, in the first list next to the console home button, choose **API Keys**.
6. Choose **Create API Key**.
7. For **Name**, type a name for the API key entry.
8. (Optional) For **Description**, type a description for the API key entry.
9. To enable the API key, select **Enabled**.
10. Choose **Save**. Make a note of the key displayed in **API key**.
11. For **API Stage Association**, for **Select API**, choose the name of the API.
12. For **Select stage**, choose the name of the stage.
13. Choose **Add**, and then choose **Save**.
14. Deploy or redeploy the API for the effect to take place.
15. Callers must now add to each call a custom header named `x-api-key`, along with the value of the API key. For example, if the API key value is `bkayZOMvuy8aZOhIgxq94K9Oe7Y70Hw55`, the custom header would be as follows:

```
x-api-key: bkayZOMvuy8aZOhIgxq94K9Oe7Y70Hw55
```

Note
In addition to, or instead of, enabling an API key, you can restrict access to certain IAM users only. For instructions, see Configure How a User Calls an API Method (p. 65).

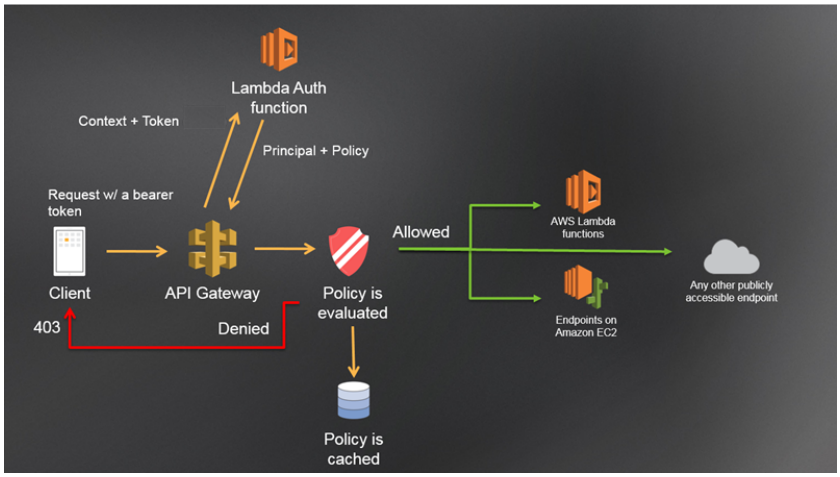
Use Amazon API Gateway Custom Authorizers

Topics

Amazon API Gateway Custom Authorization Overview

With Amazon API Gateway custom authorization, you can control access to your APIs using bearer token authentication strategies, such as OAuth or SAML. To do so, you provide and configure a custom authorizer, a Lambda function you own, for API Gateway to use to authorize the client requests for the configured APIs.

When an API request is made, API Gateway verifies whether a custom authorizer is configured for the API. If so, API Gateway calls the Lambda function, supplying the authorization token extracted from a custom request header. You use this Lambda function to implement various authorization strategies, such as JSON Web Token (JWT) verification and OAuth provider callout, to return IAM policies that authorize the request. If the returned policy is invalid or the permissions are denied, the API call will not succeed. For a valid policy, API Gateway caches the returned policy, associated with the incoming token and used 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.



Create the API Gateway Custom Authorizer Lambda Function

Before creating an API Gateway custom authorizer, you must first create the AWS Lambda function that implements the logic to authenticate and authorize the caller. You can do so in the Lambda console, using the code template available from the API Gateway Custom Authorizer blueprint. Or you can create one from scratch. For illustration purposes, we will explain here the creation of the Lambda function without using the blueprint.

> **Note**
> The custom authorizer Lambda function presented here is for illustration purposes. In production code, you should follow the API Gateway Custom Authorizer blueprint to implement your authorizer Lambda function.

When creating the Lambda function for your API Gateway custom authorizer, you will be asked to assign an execution role for the Lambda function if it calls other AWS services. For the following example, the basic `AWSLambdaRole` will suffice. For more involved use cases, follow the instructions to grant permissions in an execution role for the Lambda function.

In the code editor of the Lambda console, enter the following Node.js code.

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

exports.handler = function(event, context) {
    var token = event.authorizationToken;
    // Call oauth provider, crack jwt token, etc.
    // In this example, the token is treated as the status for simplicity.

    switch (token) {
        case 'allow':
            context.succeed(generatePolicy('user', 'Allow', event.methodArn));

            break;
        case 'deny':
            context.succeed(generatePolicy('user', 'Deny', event.methodArn));
            break;
        case 'unauthorized':
            context.fail("Unauthorized");
            break;
        default:
            context.fail("error");
    }
};

var generatePolicy = function(principalId, effect, resource) {
    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;
    }
    return authResponse;
}
```

The preceding Lambda function returns an Allow IAM policy on a specified method if the request's authorization token contains an `'allow'` value, thereby permitting a caller to invoke the specified method. The caller receives an 200 OK response. The function returns a Deny policy against the specified method if the authorization token has a `'deny'` value, thus blocking the caller from calling the method. The client will receive a 403 Forbidden response. If the token is `'unauthorized'`, the client will receive a 401 Unauthorized response. If the token is `'fail'` or anything else, the client will receive a 500 Internal Server Error response. In both of the last two cases, the calls will not succeed.

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.

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 a Custom Authorizer (p. 206) and Output from a Custom Authorizer (p. 207).

Input to an Amazon API Gateway Custom Authorizer

When a custom authorizer is enabled on an API method, you must specify a custom header for the method caller to pass the required authorization token in the initial client request. Upon receiving the request, API Gateway extracts the token from the custom header as the input `authorizationToken` parameter value into the Lambda function and calls the custom authorizer with the following request payload.

```
{
    "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 payload type. Currently, the only valid value is the `TOKEN` literal. The *`<caller-supplied-token>`* originates from the custom 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 custom authorizer configuration.

For the custom authorizer shown in the preceeding 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/"
}
```

Output from an Amazon API Gateway Custom Authorizer

The custom authorizer's Lambda function must return a response that includes the principal identifier (`principalId`) and a policy document containing a list of policy statements. The following shows an example of a response.

```
{
  "principalId": "xxxxxxxx", // The principal user identification associated
with the token send by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-api:<regionId>:<accountId>:<ap
pId>/<stage>/<httpVerb>/[<resource>/<httpVerb>/[...]]"
      }
    ]
  }
}
```

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).

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 back end. For more information, see Accessing the $context Variable (p. 102).

For information about valid policies for calling an API, see Statement Reference of IAM Policies for Executing API in API Gateway (p. 192).

The following shows example output from the example custom authorizer. The example output contains a policy statement to block (`Deny`) calls to the `GET` method in an API (`ymy8tbxw7b`) 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:ymy8tbxw7b/*/GET/"
      }
    ]
  }
}
```

Configure Custom Authorizer Using the API Gateway Console

After you create the Lambda function and verify that it works, you can configure the API Gateway Custom Authorizer in the API Gateway console.

Enable a Custom Authorizer on API Methods

1. Sign in to the API Gateway console.
2. Create a new or select an existing API and choose **Authorizers**.
3. Choose **Create**, select **Custom Authorizer**, and do the following:

   - In **Lambda region**, select the region where you upload your custom authorizer's Lambda function.
   - In **Lambda function**, select the Lambda function for your custom authorizer.

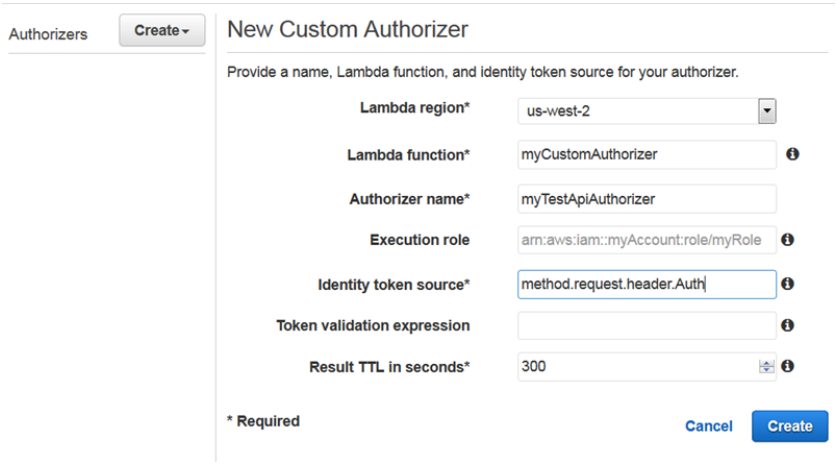
     Note
     You must first create a custom authorizer Lambda function in the region for it to be available in the drop-down list.

   - In **Authorizer Name**, enter a name for your new custom authorizer.
   - Leave **Execution role** blank to let the API Gateway console to set a resource-based policy to grant API Gateway permissions to invoke the authorizer Lambda function or 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. 144).
   - In **Identity token source**, type the mapping expression for your authorizer's custom header.

     Note
     The custom header mapping expression is of the `method.request.header.<name>` format, where `<name>` is the name of a custom authorization header submitted as part of the client request. In the following example, this custom header name is `Auth`.

   - In **Token validation expression**, you can optionally provide a RegEx statement for API Gateway to validate the input token before calling the custom authorizer Lambda function. This helps you avoid or reduce the chances of being charged for processing invalid tokens.
   - In **Result TTL in seconds**, you can change or use the default (`300`) value to enable caching (`>0`) or disable caching (`=0`) of the policy returned from the Lambda function.

     Note
     The policy caching uses a cache key generated from the supplied token for the targeted API and custom authorizer in a specified stage. 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.

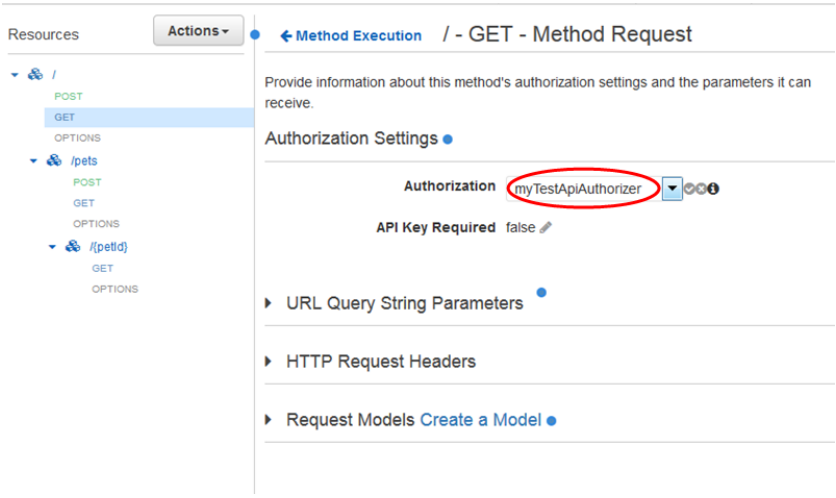


4. If you choose to let the API Gateway console to set the resource-based policy, the **Add Permission to Lambda Function** dialog will be displayed. Choose **OK**. After the custom authorization is created, you can test it with appropriate authorization token values to verify that it works as expected.

This completes the procedure to create a custom authorization. The next procedure shows how to configure an API method to use the custom authorizer.

Configure an API Method to Use the Custom 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 **Authorization Settings**, expand the **Authorization** drop-down list to select the custom authorizer you just created (**myTestApiAuthorizer**), and then choose the checkmark icon to save the choice.



4. Optionally, while still on the **Method Request** page, choose **Add header** if you also want to pass the custom authorization header to the back end. In **Name**, type a custom header name that matches the header mapping expression you used when you created the custom authorization, and then choose the checkmark icon to save the settings.

5. Choose **Deploy API** to deploy the API to a stage. Make a note of the **Invoke URL** value. You will need it when calling the API.

Call an API Using API Gateway Custom Authorization

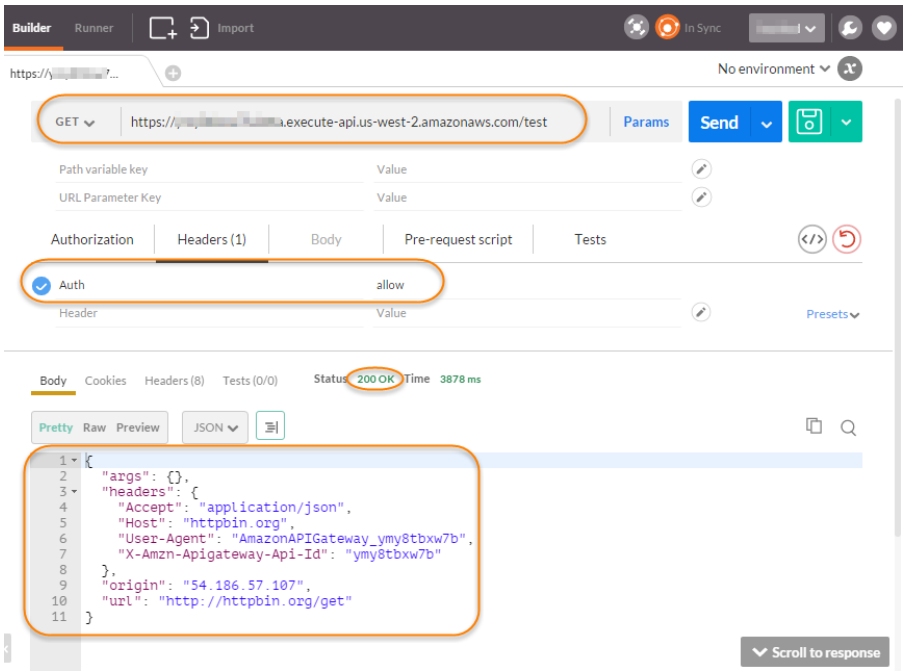
After you configure your API to use the custom authorizer, you or your customers can call the API using the custom authorizer. Because it involves submitting a custom authorization token header in the requests, you need a REST client that supports this. In the following examples, API calls are made using the Postman Chrome App.

> **Note**
> When calling an authorizer-enabled method, API Gateway will not log the call to CloudWatch if the required token is not set, null or invalidated by the specified **Token validation expression**.

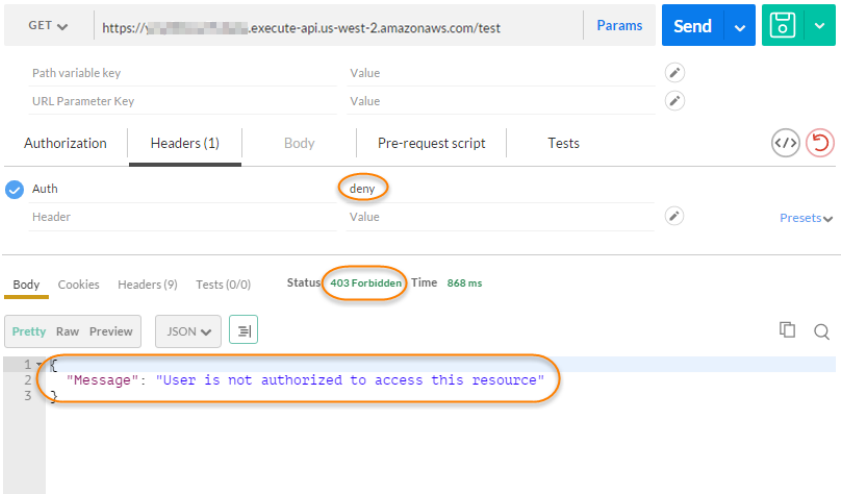
Calling an API with Custom Authorization Tokens

1. Open the **Postman Chrome App**, choose the **GET** method and paste the API's **Invoke URL** into the adjacent URL field.

   Add the custom authorization token header and set the value to `allow`. Choose **Send**.

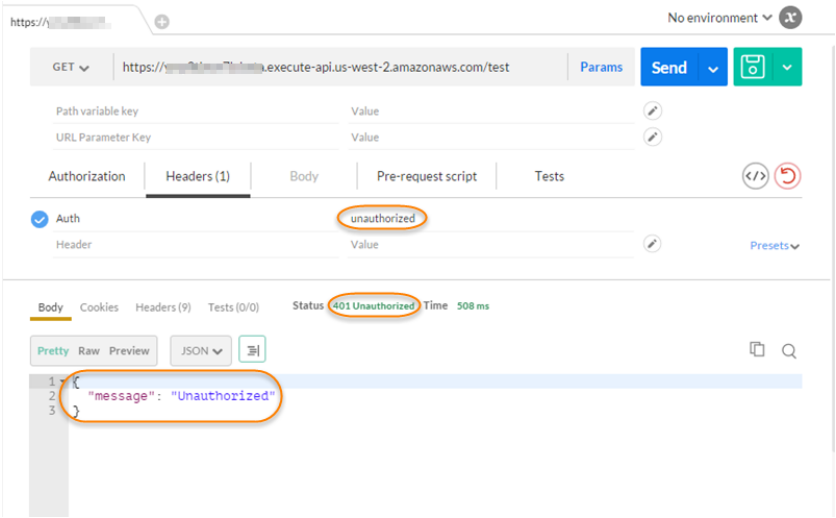


   The response shows that the API Gateway custom authorizer returns a **200 OK** response and successfully authorizes the call to access the HTTP endpoint (http://httpbin.org/get) integrated with the method.
2. Still in Postman, change the custom authorization token header value to `deny`. Choose **Send**.



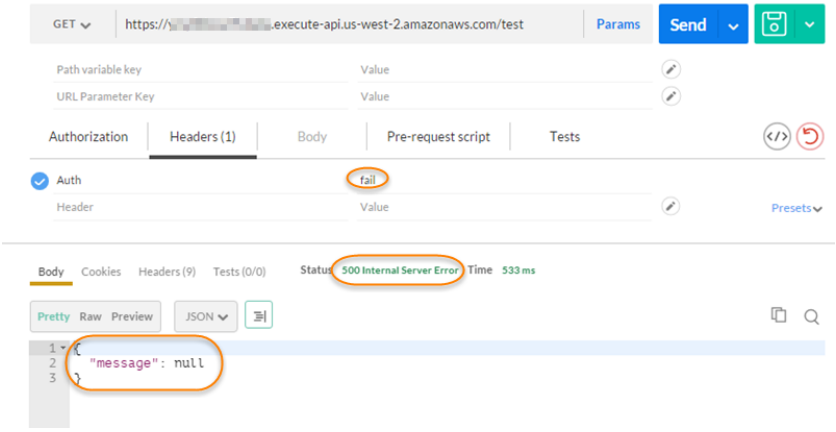
The response shows that the API Gateway custom authorizer returns a **403 Forbidden** response without authorizing the call to access the HTTP endpoint.

3. In Postman, change the custom 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 custom 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.

Authenticate API Clients with Amazon Cognito Your User Pool

In addition to using IAM roles and policies (p. 188) or custom authorizers (p. 204), you can also use a user pool in Amazon Cognito to control who can access your API in API Gateway. A user pool serves as your own identity provider to maintain a user directory. It supports user registration and sign-in, as well as provisioning identity tokens for signed-in users.

A user pool is integrated with an API as a method authorizer. When calling the methods with such an authorizer enabled, an API client includes in the request headers the user's identity token provisioned from the user pool. API Gateway then validates the token to ensure it belongs to the configured user pool and authenticates the caller before passing the request to the back end.

To integrate an API with the Amazon Cognito identity provider, you, as an API developer, create and own a user pool, create an API Gateway authorizer connected to the user pool, and enable the authorizer on selected API methods. You must also distribute to your API client developers the user pool ID, a client ID, and possibly the associated client secret that are provisioned from the user pool. The client will need this information to register users with the user pool, to provide the sign-in functionality, and to have the user's identity token provisioned from the user pool.

In this section, you will learn how to create a user pool, how to integrate an API Gateway API with the user pool, and how to invoke an API integrated with the user pool.

Topics

- Create a User Pool (p. 212)
- Integrate an API with a User Pool (p. 213)
- Call an API Integrated with a User Pool (p. 214)

Create a 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
Make note of the user pool ID, client ID and the client secret, if selected. The client will need to provide them to Amazon Cognito for the user to register with the user pool, to sign in to the user pool, and to get an identity token to be included in requests to call API methods configured with the user pool. Also, you will need to specify the user pool name when you configure the user pool as an authorizer in API Gateway, as described next.

Integrate an API with a User Pool

To integrate your API with a user pool, you must create in API Gateway a user pool authorizer connected to the user pool. The following procedure walks you through the steps to do this using the API Gateway console.

To create a user pool authorizer 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** and then choose **Cognito User Pool Authorizer**.
4. To configure this authorizer:
    a. Choose a region for **Cognito region**.
    b. For **Cognito User Pool**, choose an available user pool.
    c. The **Authorizer name** field will be automatically populated with the chosen user pool name. However, you can customize it if you want to.
    d. The **Identity token source** field will be set to `method.request.header.Authorization` by default. However, you can customize it if you want to. Using the default, `Authorization` will be the name of the incoming request header to contain an API caller's identity token.
    e. Optionally, type a regular expression in the **App client ID regex** field to validate client IDs associated with the user pool.
    f. Choose **Create** to finish integrating the user pool with the API.
5. Having created the authorizer, you can, optionally, test it by supplying an identity token provisioned from the user pool.

To enable a user pool authorizer on methods

1. Choose (or create) a method of your API.
2. Choose **Method Request**.
3. Under **Authorization Settings**, choose the edit icon by the **Authorization** field.
4. Choose one of the available **Amazon Cognito User Pool authorizers** from the drop-down list.
5. Choose the check-mark icon to save the settings.
6. Repeat these steps for other methods of your choosing.
7. If needed, choose **Integration Request** to add `$context.authorizer.claims['property-name']` or `$context.authorizer.claims.property-name` expressions in a body-mapping template to pass the specified identity claims property from the user pool to the back end. For simple property names, such as `sub` or `custom-sub`, the two notations are identical. For complex property names, such as `custom:role`, the dot notation may not be used. For example, the following mapping expressions pass the claim's standard fields of `sub` and `email` to the back end:

```
{
 "context" : {
  "sub" : "$context.authorizer.claims.sub",
  "email" : "$context.authorizer.claims.email"
 }
}
```

If you have declared a custom claim field when configuring your 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']"
    }
}
```

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 creating 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 Your User Pools and Using Your Amazon Cognito User Pool for iOS.

To call an API 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 configured with a user pool authorizer, supplying the unexpired token in the `Authorization` header or another header of your choosing.
5. When the token expires, repeat Step 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 Back End

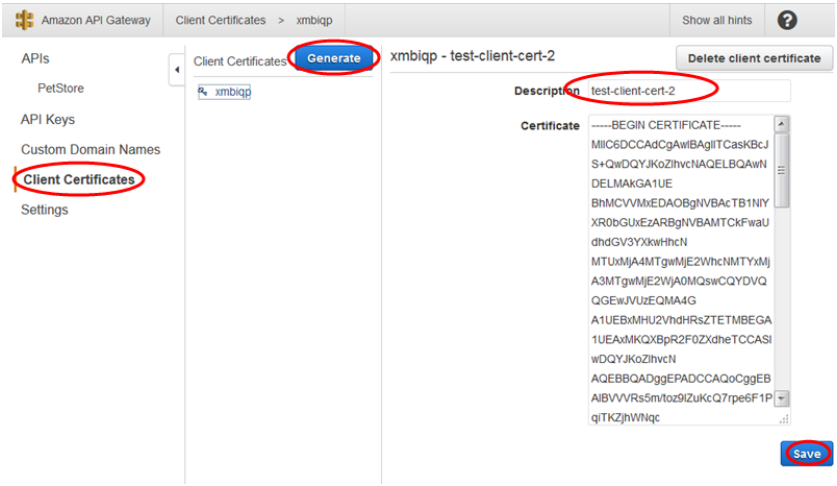
You can use API Gateway to generate an SSL certificate and use its public key in the back end to verify that HTTP requests to your back-end system are from API Gateway. This allows your HTTP back end to control and accept only requests originating from Amazon API Gateway, even if the back end is publicly accessible.

The API Gateway-generated SSL certificates are self-signed and only the public key of a certificate is visible in the API Gateway console or through the APIs.

Topics

Generate a Client Certificate Using the API Gateway Console

1. In the main navigation pane, choose **Client Certificates**.
2. From **Client Certificates**, choose **Generate**.
3. Optionally, For **Description**, enter a short descriptive title for the generated certificate. API Gateway generates a new certificate and returns the new certificate GUID, along with the PEM-encoded public key.
4. Choose the **Save** button to save the certificate to API Gateway.

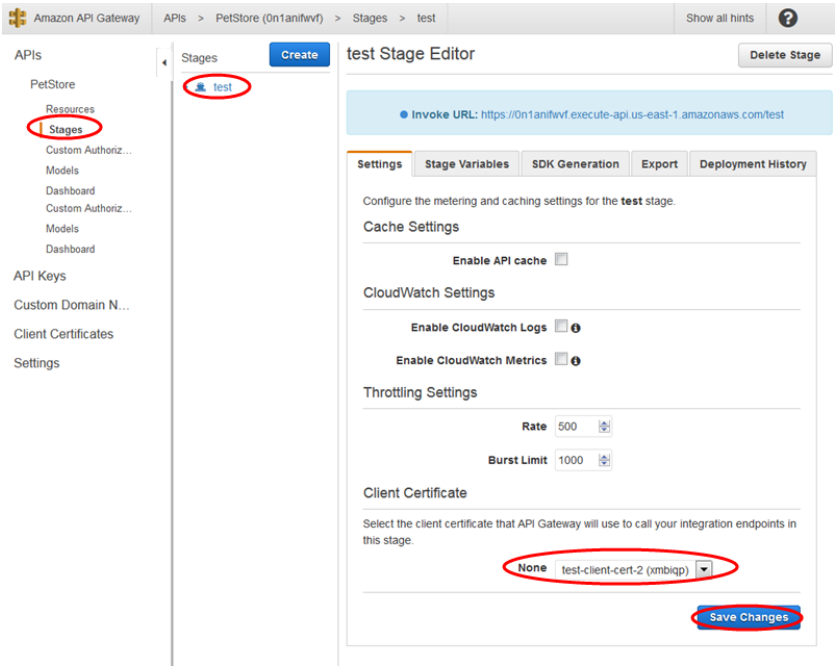


You are now ready to configure an API to use the certificate.

Configure an API to Use SSL Certificates

These instructions assume you have already completed Generate a Client Certificate Using the API Gateway Console (p. 215).

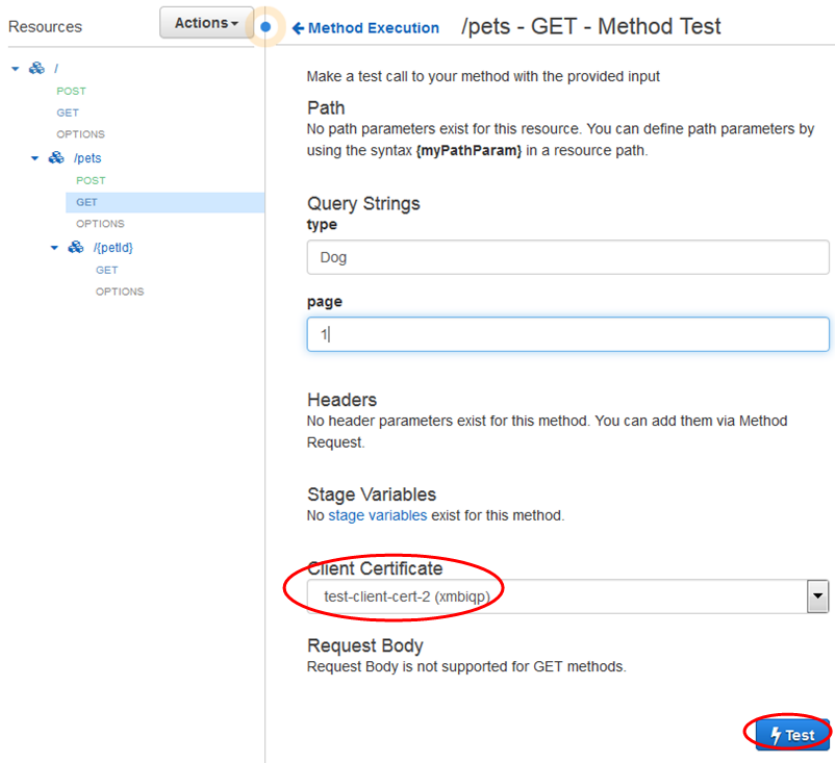
1. In the API Gateway console, create or open an API for which you want to use the client certificate. Make sure the API has been deployed to a stage.
2. Choose **Stages** under the selected API and then choose a stage.
3. In the **Stage Editor** panel, select a certificate under the **Client Certificate** section.
4. Choose the **Save Changes** button to save the settings.



After a certificate is selected for the API and saved, API Gateway will use the certificate for all calls to HTTP integrations in your API.

Test Invoke

1. Choose an API method. In **Client**, choose **Test**.
2. From **Client Certificate**, choose **Test** to invoke the method request.



API Gateway will present the chosen SSL certificate for the HTTP back end to authenticate the API.

Configure Back End to Authenticate API

These instructions assume you have already completed Generate a Client Certificate Using the API Gateway Console (p. 215) and Configure an API to Use SSL Certificates (p. 216).

When receiving HTTPS requests from API Gateway, your back end can authenticate your API using the PEM-encoded certificate generated by API Gateway, provided that the back end is properly configured. Most Web servers can be easily configured to do so.

For example, in Node.js you can use the `HTTPS` module to create an HTTPS back end and use the `client-certificate-auth` modules to authenticate client requests with PEM-encoded certificates. For more information, see HTTPS on the Nodejs.org website and see client-certificate-auth on the https://www.npmjs.com/ website.

Maintaining an API in Amazon API Gateway

Topics

View a List of APIs in API Gateway

Use the API Gateway console to view a list of APIs.

Topics

Prerequisites

- You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).

View a List of APIs with the API Gateway Console

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.

Topics

Prerequisites

- You must have deployed the API at least once. Follow the instructions in Deploying an API (p. 221).

Delete an API 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 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.

Topics

Prerequisites

- You must have methods available in API Gateway. Follow the instructions in Build an API Gateway API Step by Step (p. 14).

View a Methods List 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 **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**.

Deploying an API in Amazon API Gateway

After an API is created, you must deploy it to make it public callable. A deployment takes place in stages. A stage corresponds to a version of the API in service. In each stage, you can configure stage-level throttling settings, in addition to enabling or disabling API cache or CloudWatch logs for the API's requests and responses. If the stage-level settings are enabled, you have options to override them for individual methods. You can also define stage variables and use them to pass deployment-specific environment data to the API integration at the run time.

Topics

Deploy an API with the Amazon API Gateway Console

Prerequisites

- You must specify settings for all of the methods in the API you want to deploy. Follow the instructions in Set up Method and Integration (p. 62).

Deploy an API with the API Gateway Console

Note
If you want to change a stage in API Gateway to use a different deployment, see Change a Stage to Use a Different Deployment with the API Gateway Console (p. 222) instead.

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 deploy, choose **Resources**.
3. In the **Resources** pane, choose **Deploy API**.
4. For **Deployment stage**, do one of the following:

   - To deploy the API to an existing stage, choose the name of the stage.
   - To deploy the API to a new stage, choose **New Stage**. For **Stage name**, type the name of the stage you want to use for the deployment.

     Tip
     The stage name should be meaningful, but short enough to be easy and fast to type. Your users will specify this name as part of the URL they will use to invoke the API.

5. (Optional) For **Stage description**, type a description for the stage.
6. (Optional) For **Deployment description**, type a description for the deployment.
7. Choose **Deploy**.

Update deployment configuration with the API Gateway Console

After an API is deployed to a stage, you can, optionally, modify the deployment by updating the stage settings or stage variables. After making any changes, you must redeploy the API. The following procedure demonstrates how to accomplish with the API Gateway Console.

1. If needed, choose the **Settings** tab in the **Stage Editor** pane of the API Gateway Console.

   You can then choose to use or not use API cache, to enable or disable CloudWatch logs, to change throttling settings, or to select or deselect a client certificate.
2. If needed, choose the **Stage Variables** tab in the **Stage Editor** pane of the API Gateway Console.

   You can then choose to update the values of selected stage variables.
3. If you made any change, choose the **Save Changes** button; go back to the **Resources** window; and then choose **Deploy API** again.

   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.

Change a Stage to Use a Different Deployment with the API Gateway Console

Once you have deployed an API more than once, you can choose a specific deployment for a given stage. The following procedure shows how to do this.

1. You must have deployed to the stage at least twice. Follow the instructions in Deploy an API with the API Gateway Console (p. 221).
2. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
3. In the box that contains the name of the API with the stage you want to change, choose **Stages**.

4. Choose the stage you want to update the deployment.
5. On the **Deployment History** tab, choose the option button next to the deployment you want the stage to use.
6. Choose **Change Deployment**.

Deploy an API in Stages in Amazon API Gateway

In API Gateway, a stage defines the path through which an API deployment is accessible.

Use the API Gateway console to deploy an API in stages.

Create a Stage in API Gateway

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

Topics

Prerequisites

1. You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).
2. You must have deployed the API at least once. Follow the instructions in Deploying an API (p. 221).

Create 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, choose **Stages**.
3. Choose **Create Stage**.
4. For **Stage name**, type a name for the stage.
5. (Optional) For **Stage description**, type a description for the stage.
6. For **Deployment**, choose the date and time of the existing API deployment you want to associate with this stage.
7. Choose **Create**.

View a List of Stages in API Gateway

Use the API Gateway console to view a list of stages in API Gateway.

Topics

Prerequisites

1. You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).
2. You must have deployed the API in API Gateway at least once. Follow the instructions in Deploying an API (p. 221).

View a List of Stages 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 **Stages**.

Set Up a Stage

This section walks you through the options to set up an API deployment stage in the API Gateway console.

Topics

Prerequisites

- You must have a stage available in API Gateway. Follow the instructions in Create a Stage (p. 223).

Set Up an API Deployment 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 where you want to specify settings, choose **Stages**.
3. In the **Stages** pane, choose the name of the stage.
4. To enable a cache for the API, on the **Settings** tab, in the **Cache Settings** area, select **Enable API cache**. Then, for **Cache capacity**, choose a cache size. You can use the default for other cache settings. For information on how to set up these, . Finally, choose **Save Changes**.

   Important
   By selecting this box, your AWS account may be charged for API caching.

   Tip
   To override enabled stage-level cache settings, expand the stage under the **Stages** secondary navigation pane, choose a method. Then back in the stage editor, choose **Override for this method** 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 and other stage-level cache settings, see Enable API Caching (p. 227).
5. To generate code to call the API from Android, iOS, or JavaScript, you use the **SDK Generation** tab. For more information, see Generate an SDK for an API (p. 245).
6. To enable Amazon CloudWatch Logs for all of the methods associated with this stage of this API Gateway API, do the following:

   1. On the **Settings** tab, in the **CloudWatch Settings** area, select **Enable CloudWatch Logs**.

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 logs, but not the API- or stage level logs.

2. 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. No sensitive data will be logged unless the **Log full requests/responses data** option is selected.
3. To write entries to CloudWatch Logs that contain full API call request and response information, select **Log full requests/responses data**.
4. Choose **Save Changes**. The new settings will take effect after a new deployment.

Important
Whether you 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 this, in the secondary navigation bar, in the first list next to the console home button, choose **Settings**. Then type the ARN of the IAM role in the **CloudWatch Logging role ARN** box. 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"
    }
  ]
}
```

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. 23) section of the Call Lambda Functions Synchronously (p. 22).
For more information about CloudWatch, see the Amazon CloudWatch Developer Guide.

7. To enable Amazon CloudWatch metrics for all of the methods associated with this API in API Gateway, in the **Stage Editor** pane, on the **Settings** tab, in the **CloudWatch Settings** area, select **Enable CloudWatch metrics**, and then choose **Save Changes**. The new settings will take effect after a new deployment.

   Important
   By selecting this box, your AWS account may be charged for using CloudWatch.

   Tip
   To enable CloudWatch metrics for only some methods, clear **Enable CloudWatch metrics**. In the **Stages** pane, choose each of the methods for which you want to enable CloudWatch metrics. For each method you choose, on the **Settings** tab for the method, choose **Override for this method**, and in the **CloudWatch Settings** area, select **Enable CloudWatch metrics**. Finally, choose **Save Changes**.

   For more information about CloudWatch, see the Amazon CloudWatch Developer Guide.
8. To set a default throttle limit for all of the methods associated with this API in API Gateway, in the **Stage Editor** pane, on the **Settings** tab, in the **Throttle Settings** area, do the following, and then choose **Save Changes**:

   - For **Burst Limit**, type the absolute maximum number of times API Gateway will allow this method to be called per second. (The value of **Burst Limit** must be equal to or greater than the value of **Rate**.) The default setting is 1000 request per second.
   - For **Rate**, type the number of times API Gateway will allow this method to be called per second on average. (The value of **Rate** must be equal to or less than the value of **Burst Limit**.) The default setting is 500 request per second.

   Note

   - When creating a stage, if not supplied, API Gateway will enforce the default values of 1000 for **Burst Limit** and 500 for **Rate** in the stage settings.
   - In addition, API Gateway enforces overall account level throttling at the default values of 1000 for **Burst Limit**and 500 for **Rate**. If your require a higher level of throttling on your account, contact the AWS Support Center to request an increase.
   - API Gateway uses the token bucket algorithm, including average rate and burst size, for both account and method throttling.

9. To change the stage to use a different deployment, in the **Stage Editor** pane, on the **Change Deployment** tab, choose the option button next to the deployment you want the stage to use, and then choose **Change Deployment**.

Delete a Stage in API Gateway

Use the API Gateway console to delete a stage in API Gateway.

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.

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**.

Manage API Request Throttling

Topics

Amazon API Gateway throttles API requests to your API using the token bucket algorithm. For more information, see token bucket algorithm.

Account-Level Throttling

By default, API Gateway limits the steady-state request rates to 1000 requests per second (rps) and allows bursts of up to 2000 rps across all APIs, stages, and methods within an AWS account. If necessary, you can request an increase to your account-level limits. For more information, see API Gateway Limits (p. 279).

You can view account-level throttling limits in the API Gateway console. The console displays the default account-level settings before these settings are overridden by any customization. You can also read the account-level throttling limits by using the API Gateway REST API (p. 278).

Stage-Level and Method-Level Throttling

As an API owner, you can override the account-level request throttling limits for a specific stage or for individual methods in an API. Actual stage-level and method-level throttling limits are bounded by the account-level rate limits, even if you set the stage-level or method-level throttling limits greater than the account-level limits.

You can set the stage-level or method-level throttling limits by using the API Gateway console or by calling the API Gateway REST API (p. 278). For instructions using the console, see Set Up a Stage (p. 224).

Enable Amazon API Gateway Caching in a Stage to Enhance API Performance

Topics

Amazon API Gateway Caching Overview

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.

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.

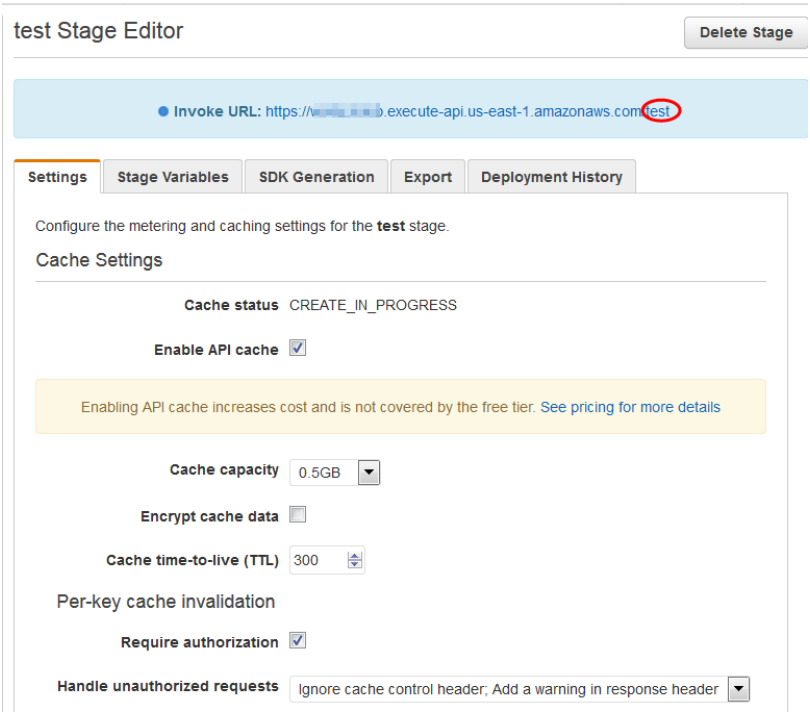
API Gateway changes caching capacity by removing the existing cache instance and recreating a new one with a modified capacity. All existing cached data is deleted.

In the API Gateway console, you configure caching in the **Settings** tab of a named **Stage Editor**.

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.



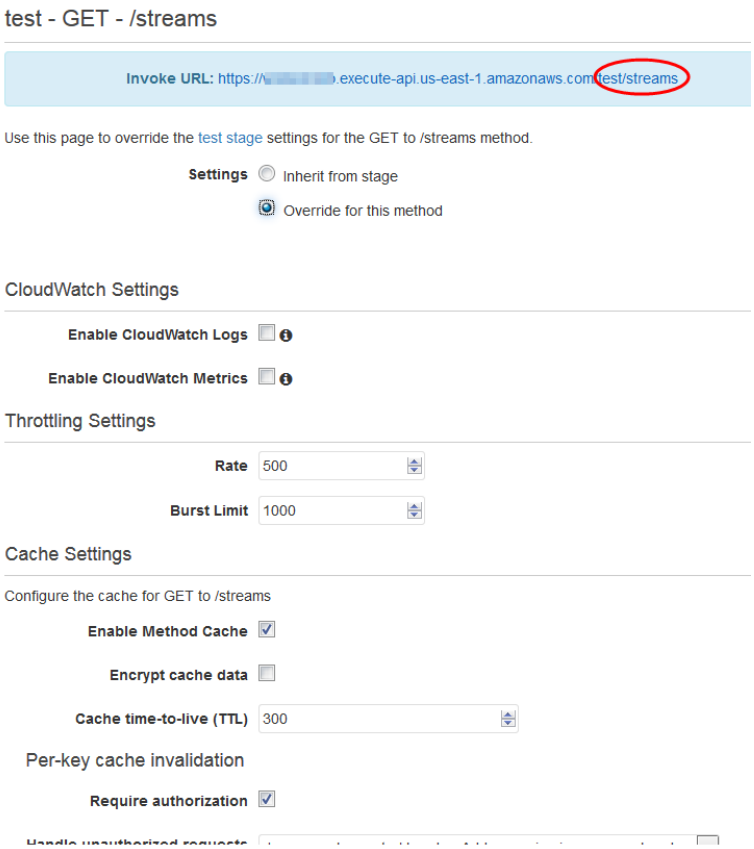
If you would like to verify if caching is functioning as expected, you have two general options:

- 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**.



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`.

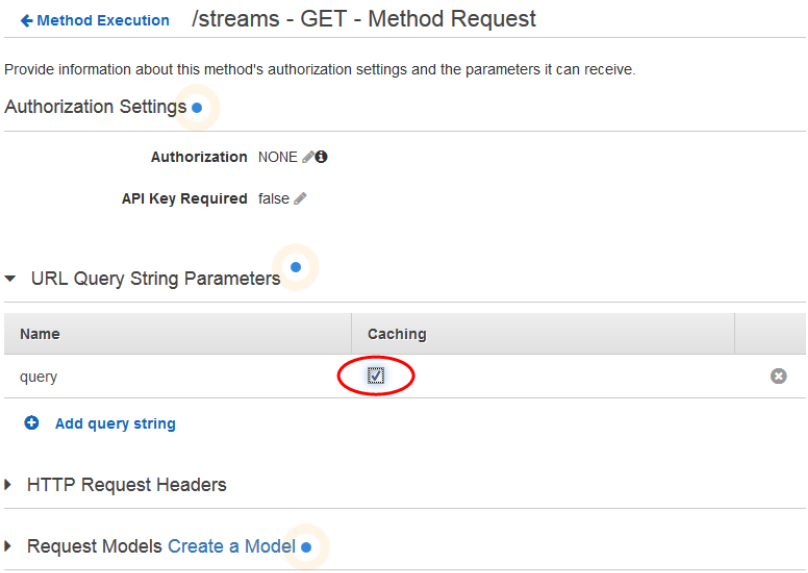
When a method or integration request takes more than one parameter, you can choose to include some or all of the parameters to create the cache key. For example, you can include only the `type` parameter in the cache key for the following request, made in the listed order within a TTL period:

```
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.



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 Cache** button under the **Stage** tab in the API Gateway console. Notice that flushing the cache will cause the responses to ensuing requests to be serviced from the back end until the cache is build 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.

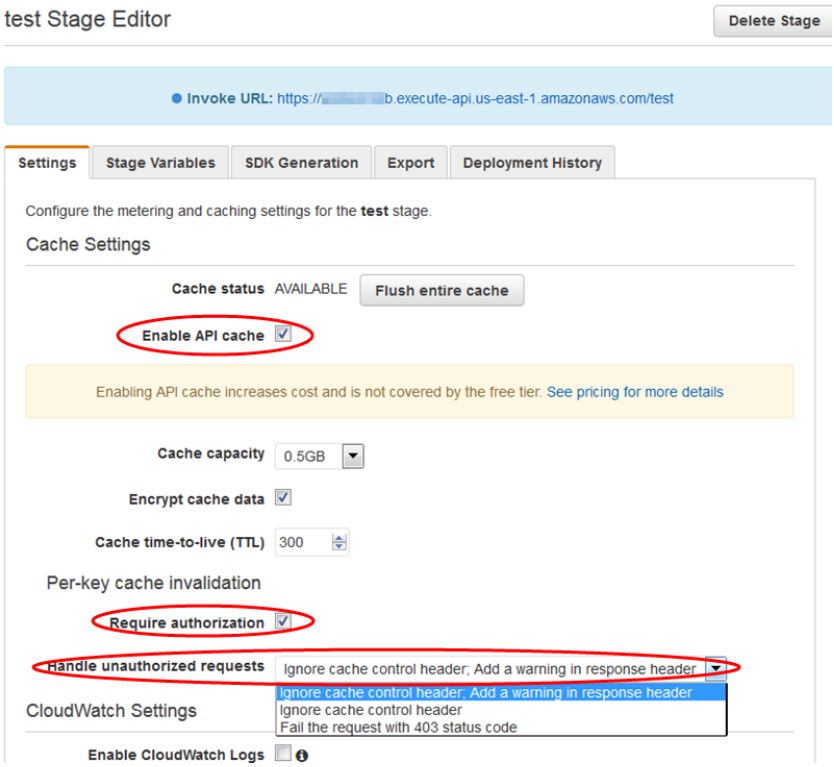
To grant permission for a caller, attach a policy of the following format to an IAM execution role for the user.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:InvalidateCache"
      ],
      "Resource": [
        "arn:aws:execute-api:region:account-id:api-id/stage-name/HTTP-VERB/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 wildcard (*) character for `account-id`, `api-id`, and other entries in the ARN value of `Resource`. For more information on how to set permissions for the API Gateway execution service, see Set IAM Permissions (p. 188)

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.



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`.

Manage API Gateway API Deployment with Stage Variables

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 back end.

For more information about mapping templates, see Request and Response Payload-Mapping Reference (p. 101).

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 back-end endpoints. For example, your API can pass a GET request as an HTTP proxy to the back-end web host (for example, `http://example.com`). In this case, the back-end 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_AC
COUNT_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

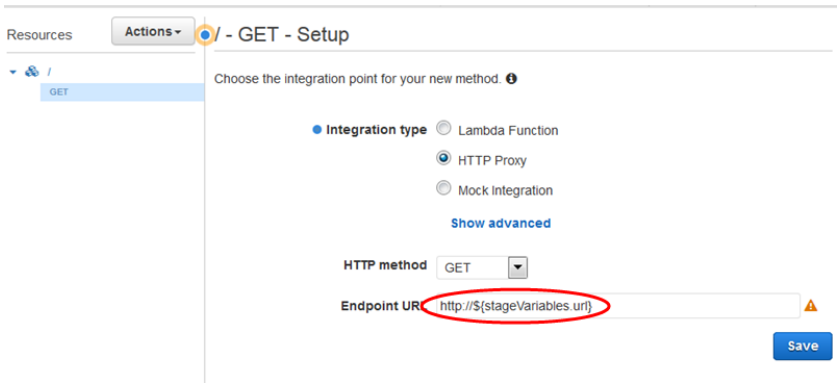
In this tutorial, you will learn how to set stage variables for two deployment stages of a sample API, using the Amazon API Gateway console.

Prerequisites

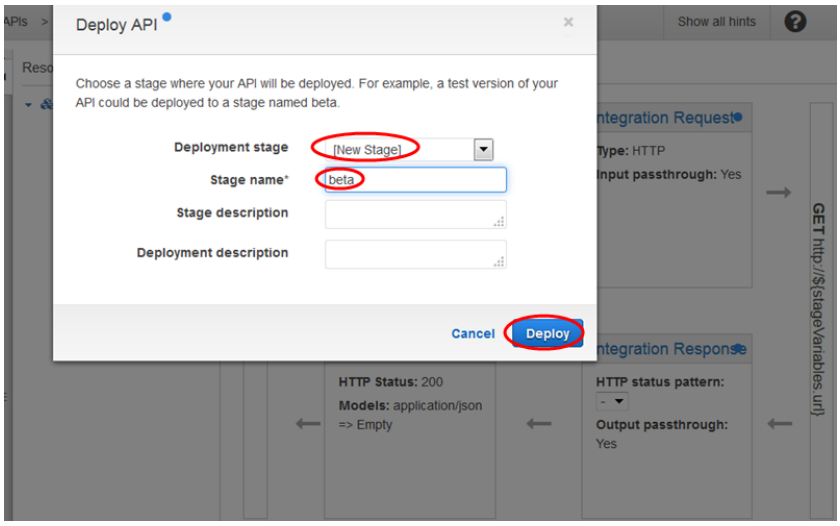
1. You must have an API available in API Gateway. Follow the instructions in Creating an API (p. 61).
2. You must have deployed the API at least once. Follow the instructions in Deploying an API (p. 221).
3. You must have created the first stage for a deployed API. Follow the instructions in Create a Stage (p. 223).

To Declare Stage Variables Using the API Gateway Console

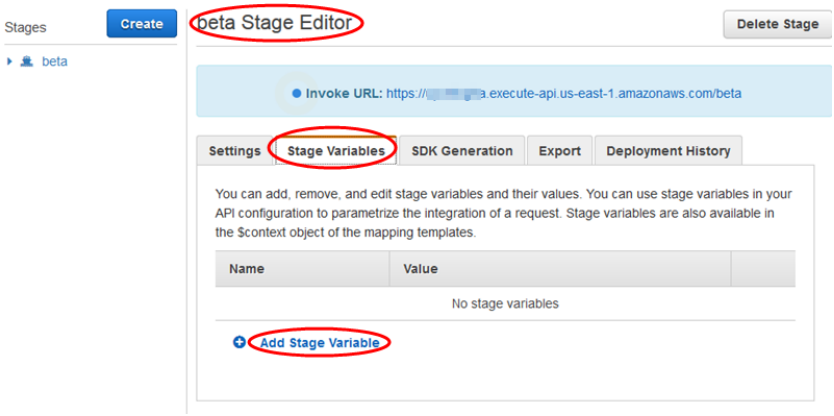
1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Create an API, create a `GET` method on the API's root resource, if you have not already done so. Set the HTTP **Endpoint URL** value as "`http://${stageVariables.url}`", and then choose **Save**.



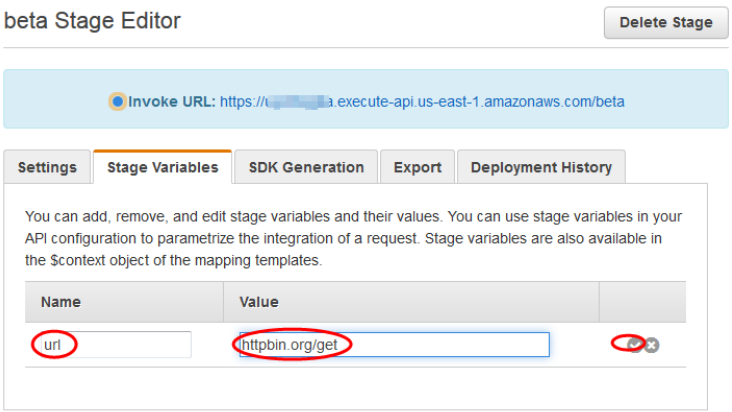
3. Choose **Deploy API**. Choose **New Stage** and enter "`beta`" for **Stage name**. Choose **Deploy**.



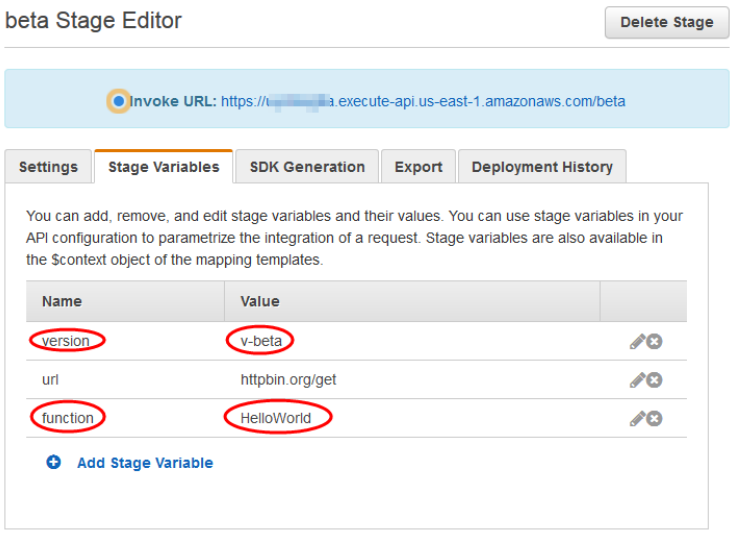
4. In the **beta Stage Editor** panel; choose the **Stage Variables** tab; and then choose **Add Stage Variable**.



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.



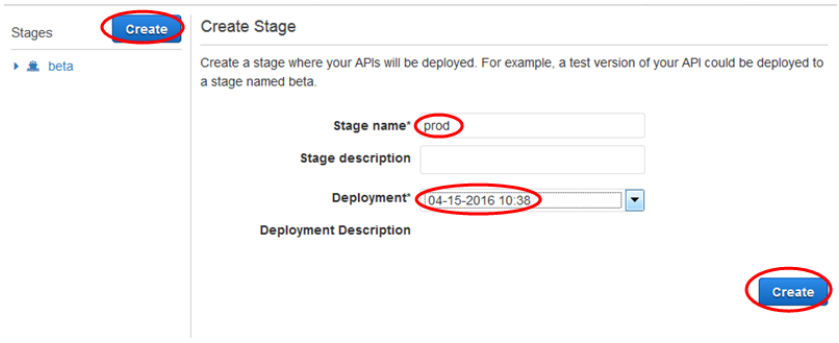
6. Repeat the above step to add two more stage variables: `version` and `function`. Set their values as "`v-beta`" and "`HelloWorld`", respectively.



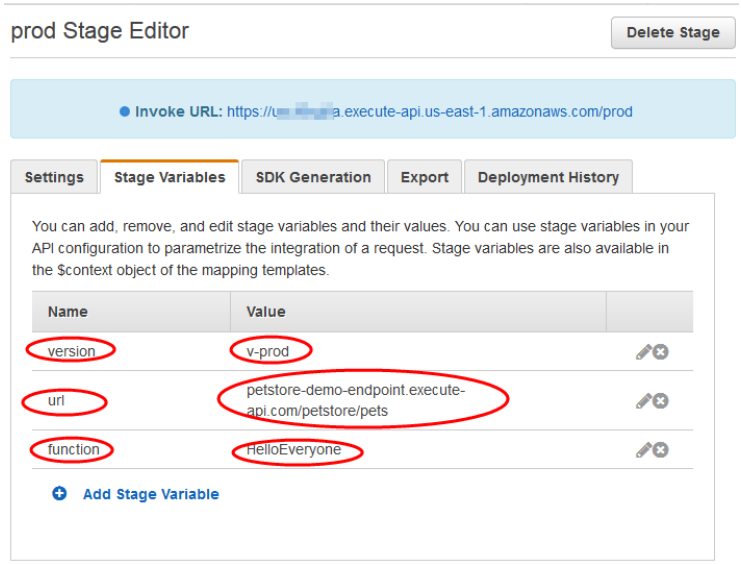
Note

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**.



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 back ends for different API deployment stages and to pass stage-specific configuration metadata into an HTTP back end as a query parameter and into a Lambda function as a payload generated in an input mapping template.

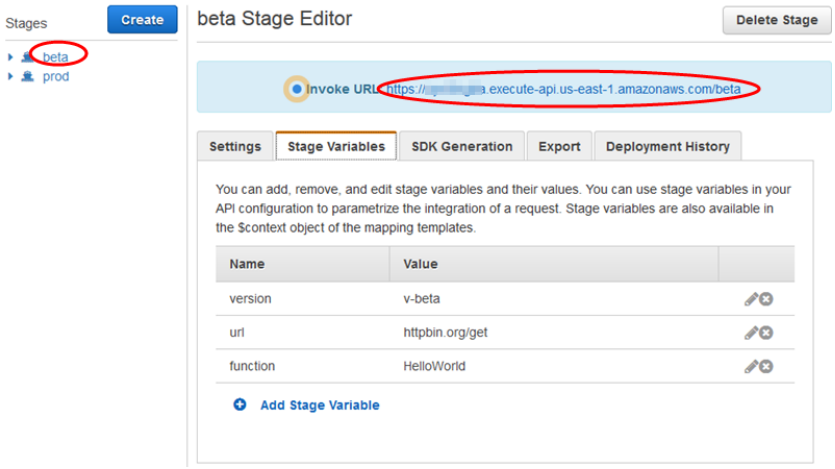
Prerequisites

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. 235).

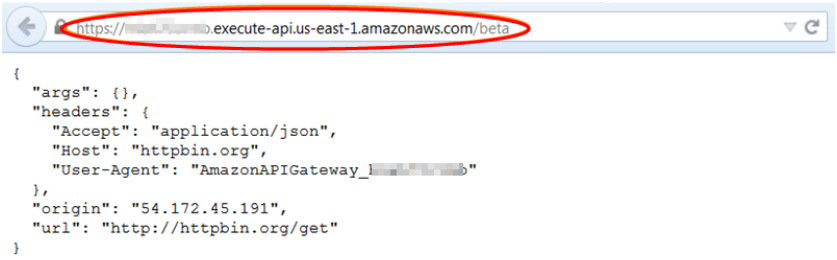
Access an HTTP endpoint through an API with a stage variable

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.

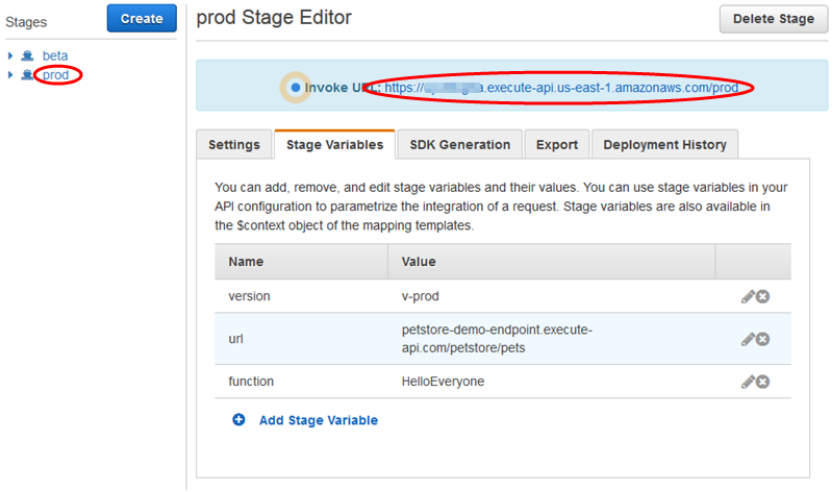
   Note
   The **Invoke URL** link points to the root resource of the API in its **beta** stage. Navigating to the URL by choosing the link calls the **beta** stage `GET` method on the root resource. If methods are defined on child resources and not on the root resource itself, choosing the **Invoke URL** 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.



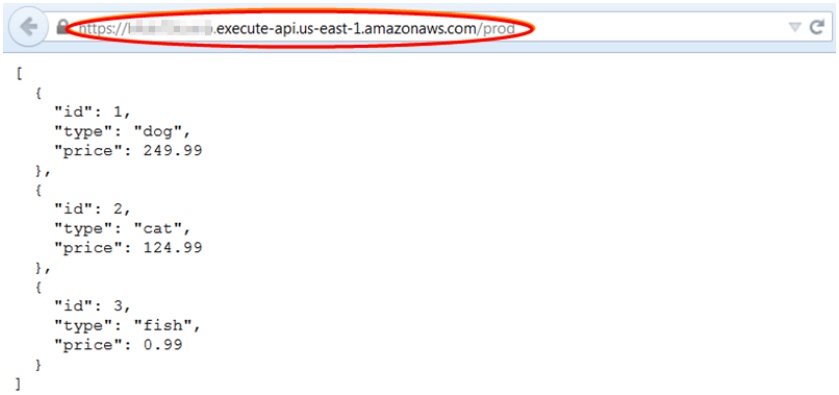
2. The response you get from the **beta** stage `GET` request is shown next. You can also verify the result by using a browser to navigate to **http://httpbin.org/get**. This value was assigned to the `url` variable in the **beta** stage. The two responses are identical.



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.



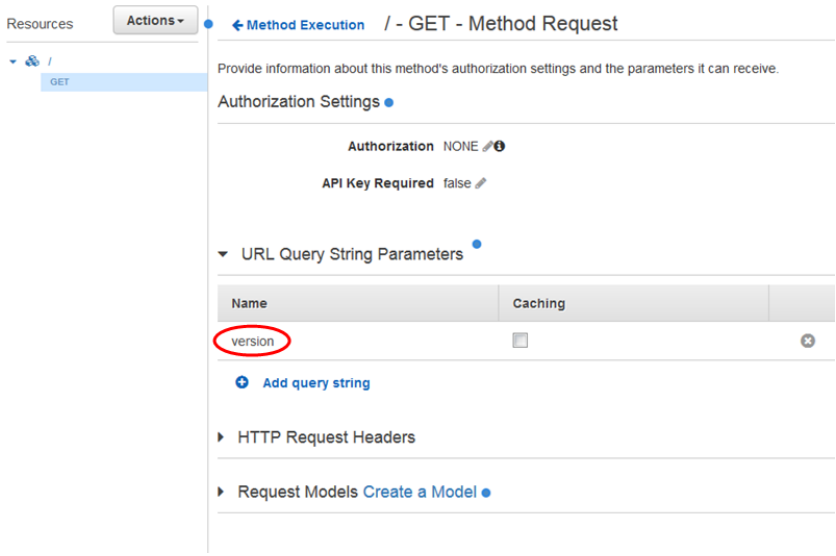
4. The response you get from the **prod** stage `GET` request is shown next. You can verify the result by using a browser to navigate to **http://petstore-demo-endpoint-execute-api.com/petstore/pets**. This value was assigned to the `url` variable in the **prod** stage. The two responses are identical.



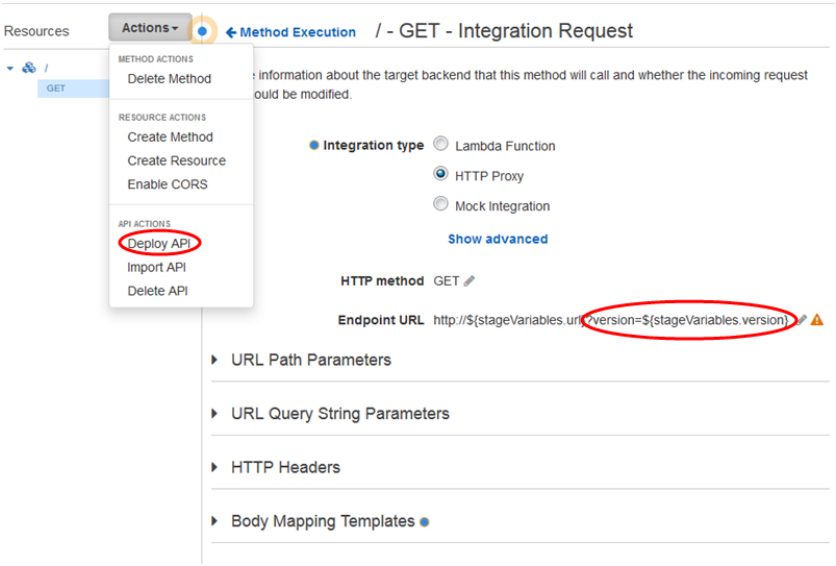
Pass stage-specific metadata to an HTTP back end via a stage variable in a query parameter expression

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. 235).

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.

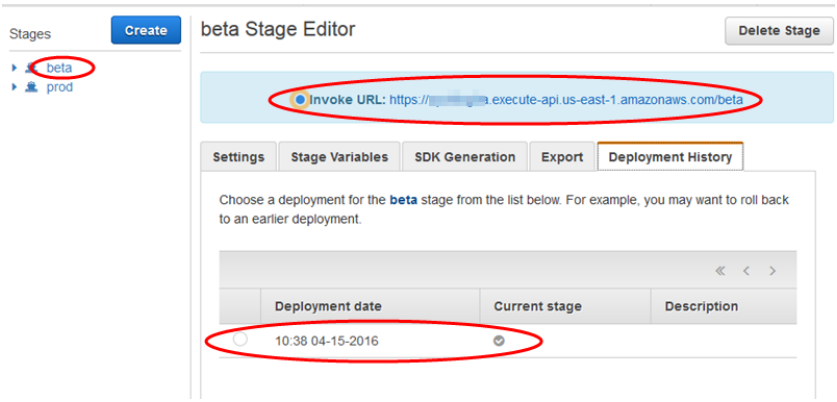


2. In **Method Execution** choose **Integration Request**. Edit the **Endpoint URL** value to append `?version=${stageVariables.version}` to the previously defined URL value, which, in this case, is also expressed with the `url` stage variable. Choose **Deploy API** to deploy these changes.



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.



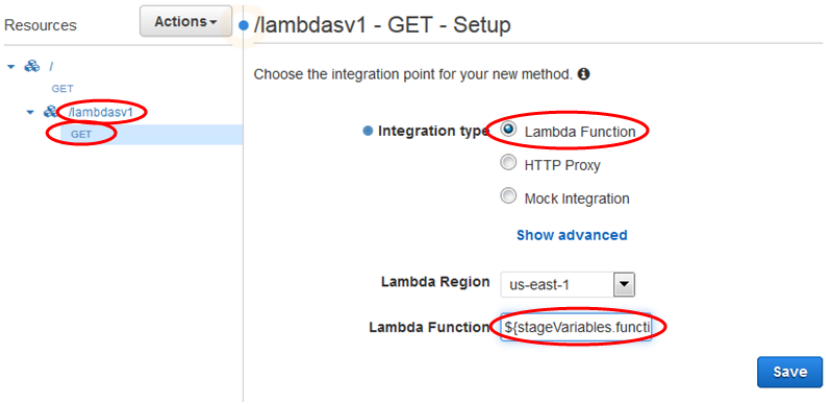
4. The response is shown next. Notice that `v-beta`, assigned to the `version` stage variable, is passed in the back end as the `version` argument.

```
{
  "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. 235).

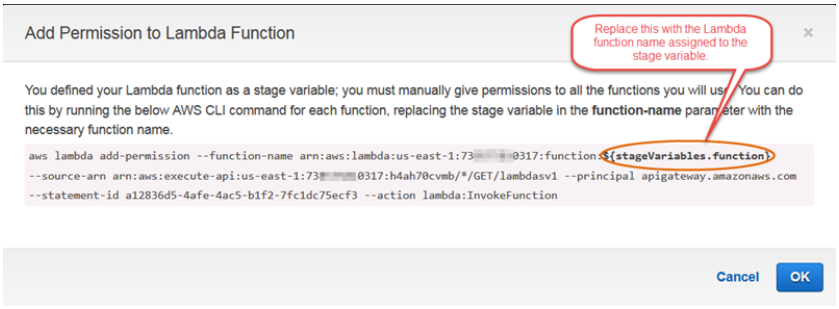
1. In the **Resources** pane, create a **/lambdasv1** child resource under the root directory, and then create a `GET` method on the child resource. Set the **Integration type** to **Lambda Function**, and in **Lambda Function**, type `${stageVariables.function}` . Choose **Save**.



Tip
When prompted with **Add Permision 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.amazon
aws.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) {
    if (event.version)
        context.succeed('Hello, World! (' + event.version + ')' );
    else
        context.succeed("Hello, world! (v-unknown)");
};
```

This implementation results in the following response.

```
"Hello, world! (v-unknown)"
```

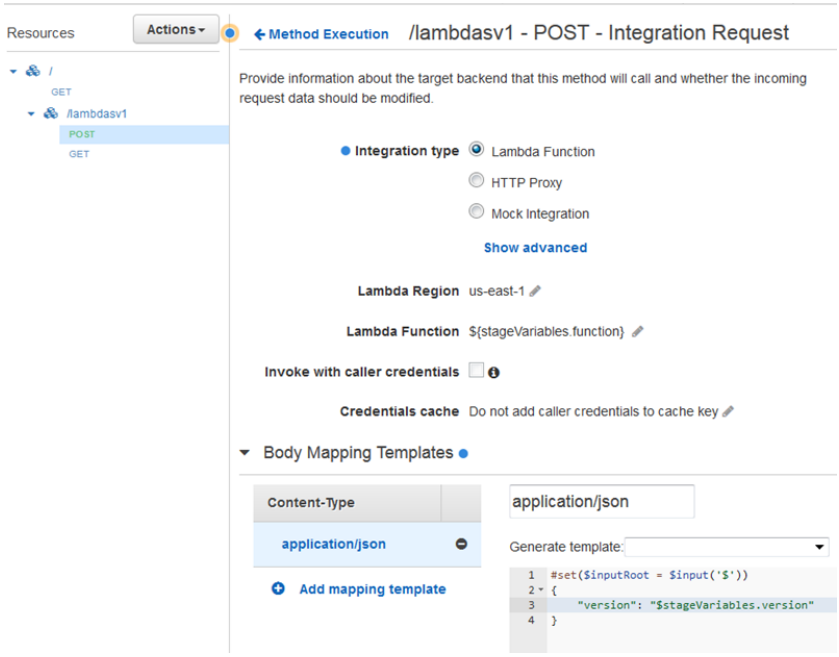
Pass stage-specific metadata to a Lambda function via a stage variable

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. 242).

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.



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>}`

Generate an SDK for an API in API Gateway

You can generate an SDK for a specific stage of an API in API Gateway. The SDK contains code you can use to call the API from Android, iOS, or JavaScript. To generate an SDK of your API, use the API Gateway console.

Topics

Prerequisites

- You must have deployed the API at least once in API Gateway. Follow the instructions in Deploying an API (p. 221).

Generate an SDK for an API 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 name of the stage.
4. On the **SDK Generation** tab, for **Platform**, choose the platform.
5. If you chose **Android**, specify the following:

   - For **Group ID**, type the unique identifier for the corresponding project. This is used in the `pom.xml` file (for example, **`com.mycompany`**).
   - For **Invoker package**, type the namespace for the generated client classes (for example, **`com.mycompany.clientsdk`**).
   - 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`**).
   - 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`**).

6. If you chose **iOS**, in the **Prefix** box, type the unique prefix for the generated classes. (For example, typing **`CLI`** will result in classes named `CLIRequestModel.h` and `CLIRequestModel.m`.)
7. Choose **Generate SDK**, and then follow the on-screen directions to download the API Gateway-generated SDK.
8. Do one of the following:

   - If you chose **Android** for **Platform**, follow the instructions in Use an API Gateway-Generated API SDK for Android (p. 247).
   - If you chose **iOS** for **Platform**, follow the instructions in Integrate an API Gateway-Generated iOS SDK into Your iOS Project (p. 248).
   - If you chose **JavaScript** for **Platform**, follow the instructions in Integrate an API Gateway-Generated JavaScript SDK into Your JavaScript Code (p. 250).

Use an API Gateway-Generated API SDK for Android

Note
These instructions assume you have already completed the steps in Generate an SDK for an API with the API Gateway Console (p. 246).

1. Extract the contents of the API Gateway-generated .zip file 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 command **mvn install** to install the compiled artifact files to your local Maven repository. This will create a `target` folder containing 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., `aws-apigateway-api-sdk-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 Andriod 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')
   ```

   and 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.
   final MyApiClient client = factory.build(MyApiClient.class);

   // Invoke a method (e.g., 'parentPath1Get(param1,body)') exposed by your
   SDK.
   // Here the method's return type is OriginalModel.
   OriginalModel output = client.parentPath1Get(param1,body);

   // You also have access to your API's models.
   OriginalModel myModel = new OriginalModel();
   myModel.setStreetAddress(streetAddress);
   myModel.setCity(city);
   myModel.setState(state);
   myModel.setStreetNumber(streetNumber);
   myModel.setNested(nested);
   myModel.setPoBox(poBox);
   ```

8. To use a Amazon Cognito credentials provider to authorize calls to your API, use the `ApiClientFactory` class to pass a set of AWS credentials by using the API Gateway-generated SDK. For example:

   ```
   // Use CognitoCachingCredentialsProvider to provide AWS credentials
   // for the ApiClientFactory
   ```

```
AWSCredentialsProvider credentialsProvider = new CognitoCachingCredentialsPro
vider(
        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");
```

Integrate an API Gateway-Generated iOS SDK into Your iOS Project

Note
These instructions assume you have already completed the steps in Generate an SDK for an API with the API Gateway Console (p. 246).

1. Extract the contents of the API Gateway-generated .zip file you downloaded earlier.
2. Import the AWS Mobile SDK for iOS into your project by using CocoaPods or Frameworks.

   To import the AWS Mobile SDK for iOS into your project by using CocoaPods, do the following:

   1. Install CocoaPods by running the command **sudo gem install cocoapods**.
   2. Copy the `Podfile` file from the extracted .zip file into the same directory as your Xcode project file. If your Xcode project file already contains a file named `Podfile`, you can simply add the following line of code to it:

      ```
      pod 'AWSAPIGateway', '~> 2.2.1'
      ```

   3. Run the **pod install** command .
   4. Use Xcode to open the `*.xcworkspace` file.
   5. Copy all of the `.h` and `.m` files from the extracted .zip file's `generated-src` directory into your Xcode project.

   To import the AWS Mobile SDK for iOS into your project by using Frameworks, do the following:

   1. Download the AWS Mobile SDK for iOS, version 2.2.1 or later.
   2. With your project already open in Xcode, press and hold the Ctrl key while choosing **Frameworks**, and then choose **Add files to "<project name>"...**.

3. In Finder, browse to and select both the `AWSCore.framework` and `AWSAPIGateway.framework` files, and then choose **Add**.
4. Open a target for your project, choose **Build Phases**, expand **Link Binary With Libraries**, choose the **+** button, and then add the following: `libsqlite3.dylib`, `libz.dylib`, and `SystemConfiguration.framework`.

3. Import the `.h` file from the API Gateway-generated SDK. For example:

```
#import "<generated header file name>"
```

4. Get the `defaultClient` from your code. For example:

```
APIGIntTestApiClient *client = [APIGIntTestApiClient defaultClient];
```

5. Use the API Gateway-generated SDK to call your API's method. For example:

```
[[client parentPath1ChildPath1Get:@"test" body:APIGOriginalModel] continue
WithBlock:^id(AWSTask *task) {
  if (task.error) {
    NSLog(@"Error: %@", task.error);
    return nil;
  }
  if (task.result) {
    APIGOriginalModel * output = task.result;
    //Do something with the output.
  }
  return nil;
  }
];
```

6. To use a Amazon Cognito credentials provider to authorize calls to your API, create an `AWSCognitoCredentialsProvider` object as the default provider for the API Gateway-generated SDK. For example:

```
AWSCognitoCredentialsProvider *creds = [[AWSCognitoCredentialsProvider alloc]
 initWithRegionType:AWSRegionUSEast1
  identityPoolId:CognitoPoolID];
AWSServiceConfiguration *configuration = [[AWSServiceConfiguration alloc]
initWithRegion:AWSRegionUSEast1 credentialsProvider:creds];
AWSServiceManager.defaultServiceManager.defaultServiceConfiguration = con
figuration;
```

7. To send an API key in your requests, set the `apiKey` property of the API Gateway-generated SDK. For example:

```
client.apiKey = @"Your API key";
```

Integrate an API Gateway-Generated JavaScript SDK into Your JavaScript Code

Note
These instructions assume you have already completed the instructions in Generate an SDK for an API with the API Gateway Console (p. 246).

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 API Gateway-generated SDK will call. For instructions, see Enable CORS for a Resource (p. 198).
3. In your web page, include references to the following scripts:

```
<script type="text/javascript" src="lib/axios/dist/axios.stan
dalone.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-tem
plate.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/sigV4Cli
ent.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/apiGatewayCli
ent.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/simpleHttpCli
ent.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 API Gateway-generated SDK by using code similar to the following:

```
var apigClient = apigClientFactory.newClient();
```

5. Call the API 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.
    });
```

6. To initialize the API Gateway-generated SDK with AWS credentials, use code similar to the following. If you use AWS credentials, 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({
  accessKey: 'ACCESS_KEY',
  secretKey: 'SECRET_KEY',
});
```

7. To use an API key with the API Gateway-generated SDK, you can 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'
});
```

Use a Custom Domain Name in API Gateway

After deploying your API, you (and the client) can invoke the API using the default root URL of the `https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com` format. To provide a simpler and more intuitive URL for your API users, you can use API Gateway to set up a custom domain name (e.g., `api.example.com`) and choose a base path (e.g., `myservice`) to present an alternative URL (e.g., `https://api.example.com/myservice`) for the API. You can also use an empty base path for an API. In this case, the API's URL is the same as the custom domain (e.g., `https://api.example.com`.)

For every API you create, API Gateway sets up an Amazon CloudFront distribution for the API. Requests with the default API URL are routed to the corresponding CloudFront distribution. Similarly, every custom domain name is backed by a CloudFront distribution. An API request with the custom domain name passes through the custom domain name's CloudFront distribution before reaching the API's CloudFront distribution. API Gateway supports custom domain names for APIs 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 enable a custom domain name, you, as the API owner, must provide a server-side SSL certificate to verify the custom domain name targeted by the client requests. You do this when setting up the domain name initially and then when renewing an expiring certificate subsequently. In addition, you must have registered the custom domain name with a domain name registrar. After setting up a custom domain name in API Gateway, you must create or update your domain name service (DNS) provider's resource record to map the custom domain name to its CloudFront distribution domain name. For the SSL certificate, you must also have obtained from a certificate authority the PEM-formatted SSL certificate body, its private key, and the certificate chain for the custom domain name. This section describes how to configure a domain name for an API, to set up the certificate for a custom domain name, to map a base path to an API, and to upload a new certificate to replace an expiring one. We will also provide general guidance, by way of examples, on how to obtain the server-side certificate and create a DNS alias record.

Topics

Prerequisites

The following steps describe how to prepare to use custom domain names in API Gateway.

To prepare to use custom domain names in API Gateway

1. Register your custom domain name. See the Accredited Registrar Directory at the ICANN website.
2. Get a PEM-encoded SSL certificate for your custom domain name from a certificate authority. For a partial list, see Third-Party Certificate Authorities at the DMOZ website.

   Here are the general steps to obtain an SSL certificate from your chosen certificate authority:

   a. Generate a private key for the certificate and save 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 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. 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
```

Set Up a Custom Domain Name for an API Gateway API

The following procedure describes how to set up a custom domain name.

To set up a custom domain name for an API Gateway API

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** in the secondary navigation pane.
4. In **Create Custom Domain Name**, specify the following:

   a. For **Domain name**, type your domain name (for example, **`api.example.com`**).
   b. For **Certificate name**, type a name for future reference (for example, **`my-example-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-----
EXAMPLEBAAKCAQEA2Qb3LDHD7StY7Wj6U2/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.

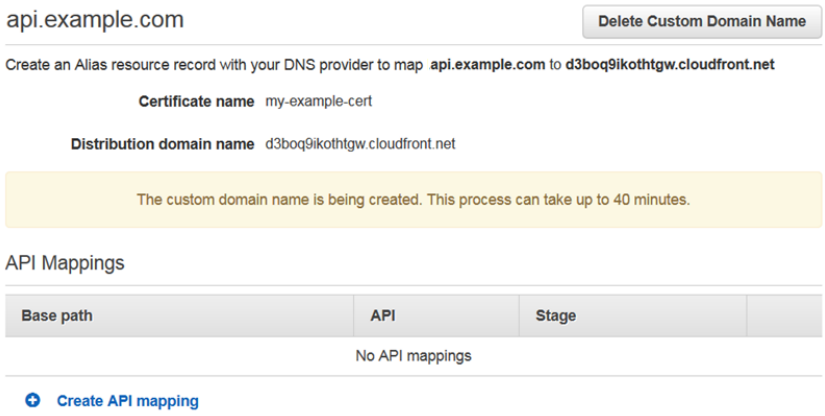
```
-----BEGIN CERTIFICATE-----
EXAMPLECA4ugAwIBAgIQWrYdrB5NogYUx1U9Pamy3DANBgkqhkiG9w0BAQUFADCB
...
```

```
8/ifBlIK3se2e4/hEfcEejX/arxbx1BJCHBvlEPNnsdw8EXAMPLE
-----END CERTIFICATE-----
```

Here is another example.

```
-----BEGIN CERTIFICATE-----
Intermediate certificate 2
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
Intermediate certificate 1
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
Optional: Root certificate
-----END CERTIFICATE-----
```

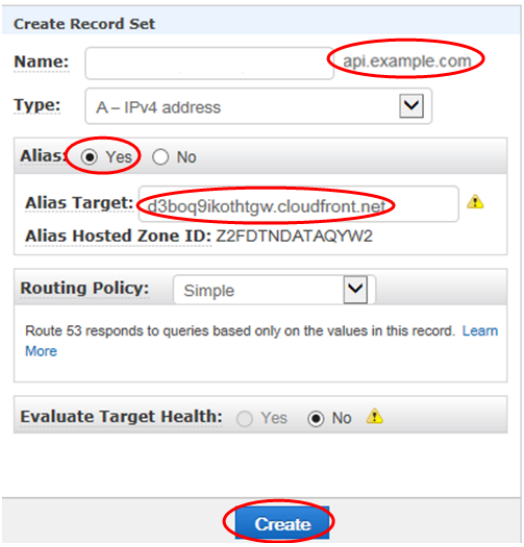
5. Choose **Save**.
6. While the new custom domain name is being created, the console displays the following information to have an alias resource record created in your DNS provider to map your custom domain name (`api.example.com`) to the API's CloudFront distribution domain name (*`distribution-id`*`.cloudfront.net`).



Make note of the CloudFront distribution's domain name shown in the output. You will need it to set the custom domain's CNAME or alias record in your DNS.

7. In this step, we will use Amazon Route 53 as an example DNS provider to show how to set up the alias record to map the custom domain to its CloudFront distribution. The instructions can be adapted to other DNS providers.

   a. Go to the Amazon Route 53 console.
   b. If necessary, register a custom domain name.
   c. Create a hosted zone.
   d. Create a record set (e.g., `api.example.com`.)
   e. Choose **Yes** for **Alias**, type the CloudFront domain name (e.g., `d3boq9ikothtgw.cloudfront.net`) in **Alias Target**, and then choose **Create**.



For most DNS providers, including Amazon Route 53, your 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 for **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.

With Amazon Route 53 you can also create an alias resource record set for your custom domain name and specify the CloudFront distribution as the alias target. This means that Amazon 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*.

Specify API Mappings for a Custom Domain Name

After you have set up a custom domain name, you must configure how individual APIs are invoked with the custom domain name. This amounts to specifying an API's URL with the given domain name. For example, if you have 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`. This involves setting up the API's base path. The first example uses an empty base path and the second example uses `myPetStore` as the base path of the API, relative to the domain name. Similarly, you can use `https://api.example.com/yourPetStore` as the `PetShop` API's URL. The base path is `yourPetShop`. Thus, base paths can be used to host multiple APIs behind a single custom domain name.

Complete the steps in Set Up a Custom Domain Name for an API Gateway API (p. 253) before setting the base path for API mappings.

To set the base path for API mappings

1. For each URL variation you want to enable, choose **Create API mapping**.
2. (Optional) For **Base path**, type the base path name that API callers must provide as part of the URL. This value must be unique for all of the mappings across a single API. Leave this blank if you do not want callers to specify a base path name after the domain name.
3. For **API**, choose the name of an available API from the selected region in your AWS account.

4. (Optional) For **Stage**, choose the name of the API's stage you want to use for this mapping. Leave this blank if you want callers to explicitly specify the stage name after any base path name.
5. Choose **Save**.

Note
To delete a mapping after you create it, next to the mapping that you want to delete, choose **Remove**.

Base Path Examples of API Mappings for a Custom Domain Name

The following examples use a custom domain name of `api.example.com`:

- Leave **Base Path** blank, specify an **API** of `MyDemoAPI`, and specify a **Stage** value of `prod` to enable calls to `https://api.example.com` to be forwarded to `https://my-api-id.execute-api.region-id.amazonaws.com/prod` (where *my-api-id* is the identifier API Gateway assigns to the API named `MyDemoAPI`).
- Leave **Base Path** blank, specify an **API** of `MyDemoAPI`, and leave **Stage** blank to enable calls to `https://api.example.com/prod` to be forwarded to `https://my-api-id.execute-api.region-id.amazonaws.com/prod` (where *my-api-id* is the identifier API Gateway assigns to the API named `MyDemoAPI`).
- Specify a **Base Path** value of `billing`, specify an **API** of `MyDemoAPI`, and leave **Stage** blank to enable calls to `https://api.example.com/billing/beta` to be forwarded to `https://my-api-id.execute-api.region-id.amazonaws.com/beta` (where *my-api-id* is the identifier API Gateway assigns to the API named `MyDemoAPI`).
- Specify a **Base Path** value of `scheduling`, specify an **API** of `MyDemoAPI`, and specify a **Stage** value of `gamma` to enable calls to `https://api.example.com/scheduling` to be forwarded to `https://my-api-id.execute-api.region-id.amazonaws.com/gamma` (where *my-api-id* is the identifier API Gateway assigns to the API named `MyDemoAPI`).

Upload and Renew an Expiring Certificate

The following steps describe how to upload and renew an expiring certificate for a custom domain name using the API Gateway console. You cannot rotate custom domain name certificates programmatically.

To upload a new certificate for a custom domain name

1. Choose **Custom Domain Names** from the API Gateway console main navigation pane.
2. Select a custom name under the **Domain Names** pane.
3. Choose **Upload**

   Note
   The upload feature will not be available when the certificate is being initialized or rotated for the selected custom domain name. However, upload for a different domain name is still available because the upload feature is independent for each custom domain name.

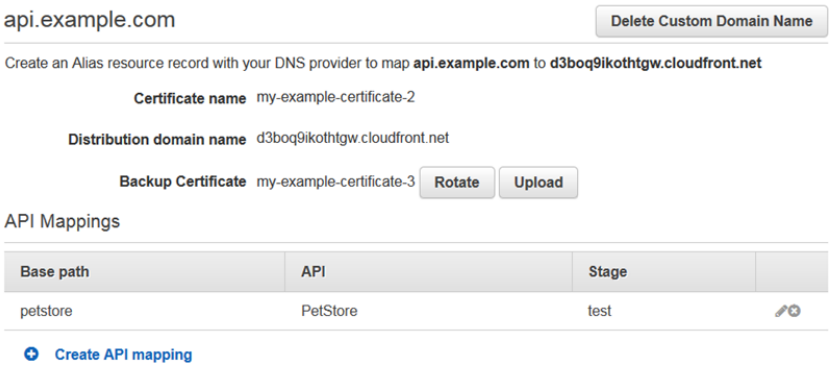
4. In **Upload Backup Certificate for *a-domain-name*** specify the following:

   - Type a name for the new certificate in **Certificate name**. The name should be different from the name of the expiring certificate.
   - Type or paste the PEM-formatted new certificate body in **Certificate body**.
   - Type or paste the PEM-formatted new certificate key in **Certificate private key**
   - Type or paste the PEM-formatted new certificate chain in **Certificate chain**.

Then, choose **Save**.

5. Choose **Rotate** to start replacing the old certificate by the new certificate.

   Note
   The certificate rotation takes up to 40 minutes to finish. The custom domain name is available during the rotation.



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.

API Gateway supports custom domain names for an API by using Server Name Indication (SNI). After a custom domain name is configured with the API, you can call the API with the custom domain name by 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.

Calling a Deployed API in Amazon API Gateway

Calling a deployed API involves submitting requests to the `execute-api` component of API Gateway. The request URL is the **Invoke URL** generated by API Gateway when the API is successfully deployed. You can obtain this invocation URL from the API Gateway console or you can construct it yourself according to the following format:

```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

If your API permits anonymous access, you can use any web browser to invoke any GET-method calls by pasting the Invoke 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 Chrome extension to test a successfully deployed API, without writing a script or a client.

Topics

Prerequisites

- You must have already deployed the API in API Gateway. Follow the instructions in Deploying an API (p. 221).

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/{resource
Path}
```

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 (p. 219).

To call the method directly from the API Gateway console, see Test a Method Using the Console (p. 259).

For more options, contact the API owner.

Test a Method Using the API Gateway Console

Use the API Gateway console to test a method.

Topics

Prerequisites

- You must specify the settings for the methods you want to test. Follow the instructions in Set up Method and Integration (p. 62).

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.

Use Postman to Test an API

Use the Postman Chrome extension is a convenient tool to test an API in API Gateway.

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 Custom authorization (p. 210).

Monitoring and Troubleshooting in API Gateway

Topics

For API execution, API Gateway automatically reports to Amazon CloudWatch your API's execution metrics on the API- and stage-levels. The metrics include statistics about caching, latency and detected errors. You can also opt in for API Gateway to send to CloudWatch method-level metrics, using the API Gateway console (p. 224) 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. For more information about CloudWatch, see Amazon CloudWatch Developer Guide.

For API management using API Gateway REST API, you can create AWS CloudTrail trails to log events involved in the API Gateway REST API calls. 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. 278). 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 Developer 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 back end.
- Monitor the **Latency** metrics to measure the overall responsiveness of your API calls.
- Monitor the **CacheHitCount** and **CacheMissCount** metrics to optimize cache capacities to achieve a desired performance.

Topics

Amazon API Gateway Dimensions and Metrics

The metrics and dimensions that API Gateway sends to Amazon CloudWatch are listed below. For more information, see Monitor API Execution with Amazon CloudWatch in the *Amazon API Gateway Developer Guide*.

API Gateway Metrics

Amazon API Gateway sends metrics under the **AWS/ApiGateway** namespace to CloudWatch once every minute.

The following metrics are available from the API Gateway service.

Metric	Description
4XXError	The number of client-side errors captured Unit: count
5XXError	The number of server-side errors captured. Unit: count
CacheHitCount	The number of requests served from the API cache. Unit: count
CacheMissCount	The number of requests served from the back end when API caching is enabled. Unit: count
Count	The number of calls to API methods. Unit: count
IntegrationLatency	The time between when API Gateway relays a request to the back end and when it receives a response from the back end. Unit: millisecond
Latency	The time between when API Gateway receives a request from a client and when it returns a response to the client. Unit: millisecond

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.

Dimension	Description
ApiName, Method, Resource, Stage	Filters API Gateway metrics for an API method of the specified API, stage, resource, and method. 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 `metricsEnabled` property to `true`. Enabling such metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing.
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

1. You must have an API created in API Gateway. Follow the instructions in Creating an API (p. 61).
2. You must have the API deployed at least once. Follow the instructions in Deploying an API (p. 221).
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 Set Up a Stage (p. 224). 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 Set Up an API Deployment Stage with the API Gateway Console (p. 224).

View API Gateway Metrics in the CloudWatch Console

To view metrics using the CloudWatch console

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

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 **CloudWatch Metrics by Category** pane, under the metrics category for API Gateway, select a metrics category, and then in the upper pane, scroll down to view the full list of metrics.

To view metrics using the AWS CLI

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/ApiGateway"
```

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 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. 263).
- **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 Developer 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 Using Events in the *Amazon CloudWatch Developer 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.

Creating and Using API Usage Plans in Amazon API Gateway

After you create, test, and deploy your APIs, you can 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 budgetary constraints.

What Is a Usage Plan?

A usage plan provides access to one or more deployed API stages with configurable throttling and quota limits enforced on individual client API keys. API callers are identified by API keys that can be generated by API Gateway or imported from external sources. The throttling prescribes the request rate limits applied to each API key. The quotas are the maximum number of requests with a given API key submitted within a specified time interval. Individual API methods can be configured 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.

How to Configure a Usage Plan?

The following steps describe 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) 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 enforce authorization of the API key in requests to the API, individual API methods must be configured to require an API key (p. 203). Setting this configuration ensures the incoming API key will be authorized according to the usage plan configuration.

Topics

Configure Usage Plans Using the API Gateway Console

To use the API Gateway console to configure a usage plan, use the following instructions.

Topics

Create and Deploy an API for Usage Plans

For instructions on how to create and deploy an API, see Creating an API (p. 61) and Deploying an API (p. 221), respectively.

Configure an API Method to Require an API Key

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

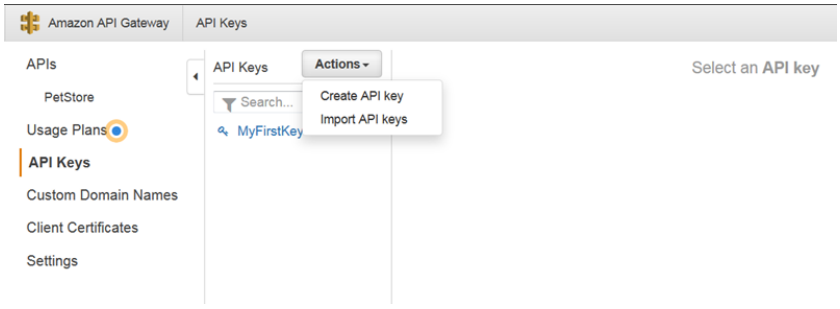
1. Sign in to the AWS Management Console and open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. In the API Gateway main navigation pane, choose **Resources**.
3. Under **Resources**, create a new method or choose an existing one.
4. Choose **Method Request**.
5. Under the **Authorization Settings** section, choose **`true`** for **API Key Required**.
6. Select the check-mark icon to save the settings.

Create an API Key

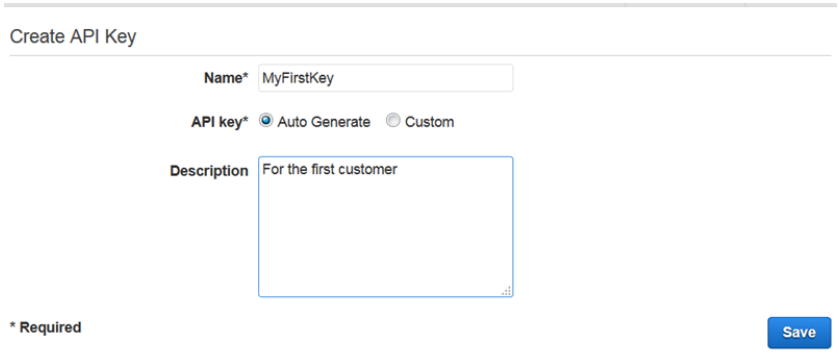
If you have already created or imported API keys for use with usage plans, you can skip this and the next procedure.

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**.



4. In **Create API Key**, do the following:
   a. Type an API key name (e.g., **`MyFirstKey`**) in the **Name** input field.
   b. Choose **Auto Generate** to have API Gateway to generate the key value or choose **Custom** to enter the key manually.
   c. Choose **Save**.



5. Repeat the preceding steps to create more API keys, if needed.

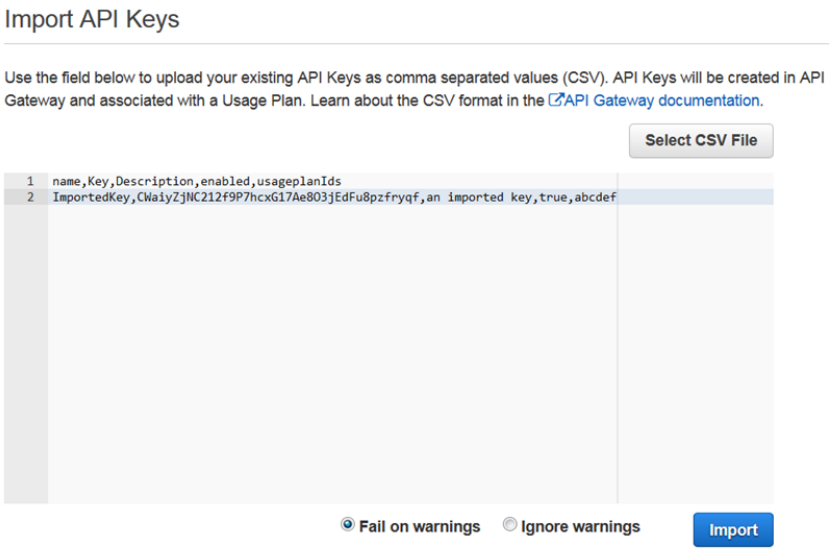
Import API Keys

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. 277).



4. Choose **Fail on warnings** to stop import when there is an error, or choose **Ignore warnings** to continue to import valid key entries when there is an error.
5. To start importing the selected API keys, choose **Import**.

Migrate to Default Usage Plans

When creating a usage plan for the first time, you are prompted with the **Enable Usage Plans** option before you can proceed further. This option creates default usage plans for every unique API stage associated with existing API keys. In the default usage plan, no throttle and quota limits are set initially, existing API keys are converted to a collection of UsagePlanKey resources, and existing API keys are converted to API stage Ids. The API will behave 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.

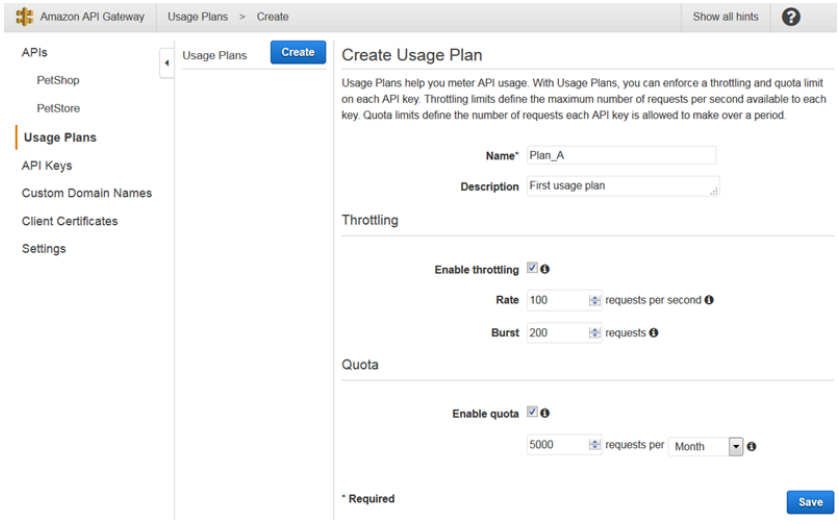
Create Usage Plans

The following procedure describes how to create a usage plan.

To create a usage plan

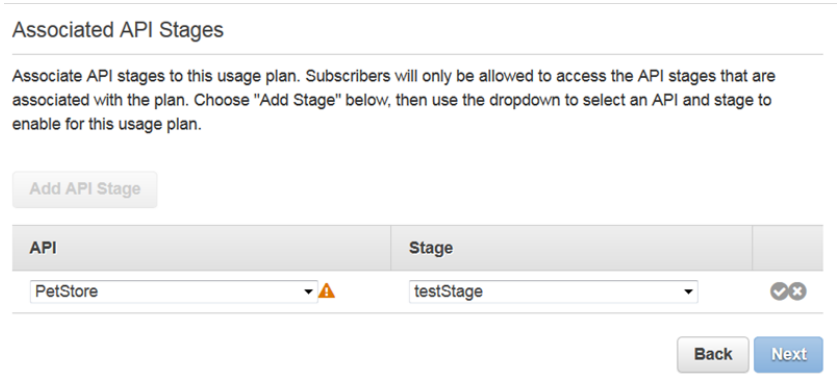
1. In the Amazon API Gateway main navigation pane, choose **Usage Plans**, and then choose **Create**.
2. Under **Create Usage Plan**, do the following:

   a. For **Name**, type a name for your plan (e.g., **`Plan_A`**).
   b. For **Description**, type a description for your plan.
   c. Select **Enable throttling** and set **Rate** (e.g., **`100`**) and **Burst** (**`e.g., 200`**).
   d. Choose **Enable quota** and set its limit (e.g., **`5000`**) for a selected time interval (e.g., **Month**).
   e. Choose **Save**.



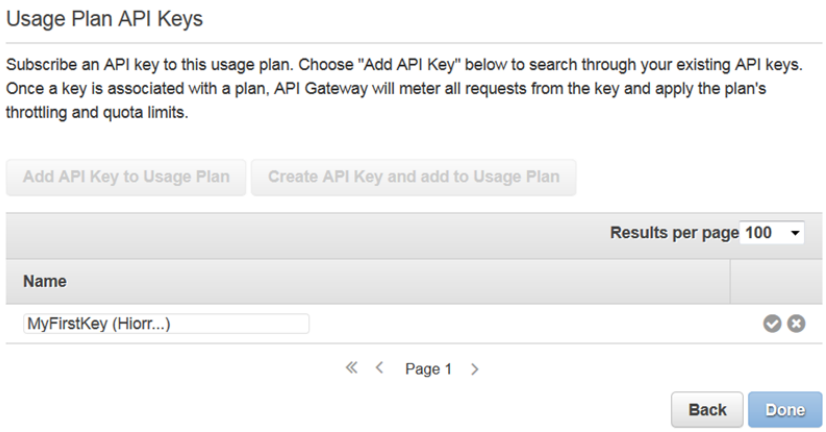
3. To add a stage to the plan, do the following in the **Associated API Stages** pane:

   a. Choose **Add API Stage**.
   b. Choose an API (e.g., **`PetStore`**) from the **API** drop-down list.
   c. Choose a stage (e.g., **`Stage_1`**) from the **Stage** drop-down list.
   d. Choose the check-mark icon to save.
   e. Choose **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 (e.g., **`MyFirstKey`**).
   c. Choose the check-mark icon to save.
   d. If desired, repeat the preceding steps to add other existing API keys to this usage plan.



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. 276)

Manage Plan Usage

Managing 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. Then choose **Usage** to view **Subscriber's Traffic** from the plan you are 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 has not yet used any allotted quotas 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**.

Configure Usage Plans Using the API Gateway REST API

To configure a usage plan using the API Gateway REST API, use the following instructions, assuming you have already created the APIs to be added to the usage plan.

Topics

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, setting `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. 277).

Migrate to Default Usage Plans

When creating a usage plan the first time, you can migrate existing API stages 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. 271).

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, specifying in the payload the name and description of the plan, associated API stages, rate limits, and quotas.

   Make note of the resultant usage plan identifier. You will 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, specifying `keyId` and `keyType` in the payload.

      To add more API keys to the usage plan, repeat the above 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 the keys are enabled for the usage plan, and, possibly, the API key names and descriptions.

      The following example of the `apikey:import` request will add three API keys (as identified by `key`, `name`, and `description`) to one usage plan (as identified by `usageplanIds`):

      ```
      POST /apikeys?mode=import&format=csv&failonwarnings=fase HTTP/1.1
      Host: apigateway.us-east-1.amazonaws.com
      Content-Type: text/csv
      Authorization: ...
      ```

```
key,name, description, enabled, usageplanIds
abcdef1234ghijklmnop8901234567, importedKey_1, firstone,  tRuE, n371pt
abcdef1234ghijklmnop0123456789, importedKey_2, secondone, TRUE, n371pt
abcdef1234ghijklmnop9012345678, importedKey_3,          , true, n371pt
```

As a result, three `UsagePlanKey` resources will be 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 usageplans:by-id to get a usage plan of a given plan Id. To see the available usage plans, call apigateway:usageplans.
2. Call usageplan:update to add a new API stage to the plan, to replace an existing API stage in the plan, to remove an API stage from the plan, or to 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, created in Build and Test an API Gateway API from an Example (p. 6). Assume 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 (e.g., `xbvxlpijch`) in a usage plan:

```
GET /testStage/pets?type=dog&page=1 HTTP/1.1
x-api-key: Hiorr45VR...c4GJc
Content-Type: application/x-www-form-urlencoded
Host: xbvxlpijch.execute-api.ap-southeast-1.amazonaws.com
X-Amz-Date: 20160803T001845Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20160803/ap-
southeast-1/execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-
date;x-api-key, Signature={sigv4_hash}
```

  Note
  You must submit this request to the `execute-api` component of API Gateway and provide the required API key (e.g., `Hiorr45VR...c4GJc`) in the required `x-api-key` header.

  The successful response returns a `200 OK` status code and a payload containing the requested results from the back end. If you forget to set the `x-api-key` header or set it with an incorrect key,

you will get a `403 Forbidden` response. On the other hand, if you did not configure the method to require an API key, you will likely to 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.

API Gateway API Key File Format

API Gateway can import API keys from external files of a comma-separated value (CSV) format and associate the imported keys with one or more usage plans. The imported file must contain the `Name` and `Key` columns. The column header names are not case-sensitive and columns can be in any order, as shown in the following example:

```
Key,name
apikey1234abcdefghij0123456789,MyFirstApiKey
```

A `Key` value must be between 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,apikey1234abcdefghij0123456789,An imported key,TRUE,c7y23b
```

When a key is associated with more than one usage plan, the `UsagePlanIds` value is a comma-separated string of the usage plan Ids enclosed with a pair of double or single quotes, as shown in the following example:

```
Enabled,Name,key,UsageplanIds
true,MyFirstApiKey,apikey1234abcdefghij0123456789,"c7y23b,glvrsr"
```

Unrecognized columns are permitted, but will be 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.

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 and Pricing

Topics

API Gateway Limits

Unless noted otherwise, the limits can be increased upon request. To request a limit increase, contact the AWS Support Center.

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	1000 request per second (rps) with a burst limit of 2000 rps.	Yes
APIs per account	60	Yes
API keys per account	500	Yes
Usage plans per account	300	Yes
Custom authorizers per API	10	Yes

Resource or Operation	Default Limit	Can Be Increased
Client certificates per account	60	Yes
Resources per API	300	Yes
Stages per API	10	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	30 second for both Lambda and HTTP integrations.	No
Payload size	10 MB	No
Number of iterations in a `#foreach ... #end` loop in mapping templates	1000	No
ARN length of a method with authorization	1600 bytes	No

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.

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	n a C e B - I d s c	
CreateRestApi	2 requests per minute (rpm) per account.	o	N
ImportRestApi	2 requests per minute per account	o	N
PutRestApi	60 requests per minutes per account	o	N
DeleteRestApi	2 requests per minutes per account	o	N
CreateDeployment	3 requests per minutes per account	o	N
UpdateAccount	3 requests per minutes per account	o	N

Action	Default Limit	n a C e B - n l d s c	
GetResources	150 requests per minutes per account	o	N
CreateResource	300 requests per minutes per account	o	N
DeleteResource	300 requests per minutes per account	o	N
CreateDomainName	2 requests per minutes per account	o	N

API Gateway Pricing

For API Gateway region-specific pricing information, see Amazon API Gateway Pricing.

Note
API caching in Amazon API Gateway is not eligible for the AWS Free Tier.

Known Issues

- Cross-account authentication is not currently supported in API Gateway. An API caller must be an IAM user of the same AWS account of the API owner.
- 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 back end, the intermediate certificate is missing from the certificate chain, or any other unrecognizable certificate-related exceptions thrown by the back end.

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:** August 11, 2016

Change	Description	Date Changed
Extending selected APIs in API Gateway as product offerings for your customers by providing one or more usage plans.	Create a usage plan in API Gateway to enable selected API clients to access specified API stages at agreed-upon request rates and quotas. For more information, see Creating and Using Usage Plans (p. 268)	August 11, 2016
Enabling method-level authorization with a user pool in Amazon Cognito	You can 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 Authenticate API Clients with Amazon Cognito Your User Pool (p. 212)	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. 263).	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 Upload and Renew an Expiring Certificate (p. 256).	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 and Test an API Gateway API from an Example (p. 6) and Build an API Gateway API Step by Step (p. 14).	April 5, 2016

Change	Description	Date Changed
Enabling the Import API feature to create a new or update an existing API from external API definitions.	With the Import API features, you can create a new API or update an existing one by uploading an external API definition expressed in Swagger 2.0 with the API Gateway extensions. For more information about the Import API, see Import an API (p. 110).	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 Request and Response Payload-Mapping Reference (p. 101).	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. 231) and Invalidate an API Gateway Cache Entry (p. 231). Improve the console experience for managing API request throttling. For more information, see Manage API Request Throttling (p. 227).	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 Custom Authorizers (p. 204).	February 11, 2016
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 and Export API (p. 109).	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 Request and Response Parameter-Mapping Reference (p. 98).	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 Manage API Gateway API Deployment with Stage Variables (p. 233).	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. 198).	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 (p. 215).	September 22, 2015

Change	Description	Date Changed
Mock integration of methods	Learn how to mock-integrate an API with Amazon API Gateway (p. 69). This feature enables developers to generate API responses from API Gateway directly without the need for a final integration back end 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 Request and Response Payload-Mapping Reference (p. 101).	August 28, 2015
Swagger integration	Use the Swagger import tool on GitHub to import Swagger API definitions into Amazon API Gateway. Learn more about Import and Export API (p. 109) 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 Request and Response Payload-Mapping Reference (p. 101).	July 18, 2015
Initial public release	This is the initial public release of the *Amazon API Gateway Developer Guide*.	July 9, 2015

AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the *AWS General Reference*.