

Using Amazon API Gateway

Overview

This document describes how to use Amazon API Gateway and is based on the AWS Developer's Guide available online (<https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>)

API Gateway Defined

Amazon API Gateway is a cloud service for creating, publishing, maintaining, monitoring, and securing REST and WebSocket Application Programming Interfaces (APIs). REST stands for **RE**presentational **State** Transfer. Characteristics of A REST service include:

- Client-Server based to improve the portability and scalability. A client can make REST API Gateway call from multiple platforms and invoke the same functionality. Scalability is provided by distributing the functionality across multiple servers to adapt to increased workloads during peak periods of demand.
- Stateless where the server does not maintain any record or history of previous actions. This results in client requests containing all of the needed information necessary.
- HyperText Transfer Protocol (HTTP)-based where data is transferred over the Web using popular methods including GET, POST, PUT, PATCH and DELETE

WebSocket API's have the following distinctive characteristics:

- Client-Server based providing portability and scalability
- Stateful providing full-duplex communication between client and server.

Clients can connect to the API Gateway via mobile, web, desktop, IoT or other devices. The request can then be routed to other Amazon services including CloudTrail, Lambda and others. The API Gateway transparently accepts and processes concurrent API calls providing the needed scalability.

Popular Use Cases

Amazon API Gateway can be used to create REST APIs that provide serverless implementations of Web pages, forms and other Web-based requests. Serverless Web sites and components can result in significant cost and environmental resources savings.

Amazon API Gateway can also be used to create WebSocket APIs providing and supporting useful client-server applications such as chat rooms, and real-time events, alerts and notifications.

API Gateway Concepts

Both REST and WebSocket API's can be thought of a collection of resources, methods or routes that are integrated with backend HTTP endpoints and are deployed via one or more stages. Stages are logical name referencing the lifecycle state of the API. Typical lifecycle states might include alpha, beta, dev, test, QA, prod and others. The stage name combined with API ID uniquely identify the API. This combination allows for concurrent testing of multiple stages before deploying to production.

An API endpoint or host URL has the form of:

```
{api-id}.execute-api.{region}.amazonaws.com
```

For example, the following is a possible API endpoint in the us-east-1 region with a stage of "prod".

<https://dsfnw5eo5s.execute-api.us-east-1.amazonaws.com/prod/>

API Keys are alphanumeric strings that can be used to identify app developers using APIs.

Key to understanding the inner workings of API gateways is to be comfortable with HTTP request and response operations. Requests are sent by the client to trigger an action on the server. Requests include an HTTP method such as [Get](#) or [Post](#). Responses are the corresponding answer from the server. They typically include a protocol version (e.g. HTTP 1.1) and a status code.

Common status codes include:

- 200 – OK indicating the request has succeeded
- 302 - Found indicating a redirect response where the resource requested has been temporarily moved to the URL
- 404 - Not Found indicating an error where the server can't find the requested resource.

Within the API Gateway Console, resources display available request and response options. In addition to the Method (e.g. Get) or Route request and responses, Integration request and responses are available. The integration request maps the body to the formats required by the backend. For example, if a Lambda function is used in the back end this is where that integration takes place. The integration response maps the responses and payload received from the backend (e.g. Lambda) to the response returned to a client. With the Integration components, the round trip from client request to backend responses are possible.

Both REST API's and WebSocket API's have similar components. For WebSocket, instead of methods, routes are used with route request and route response functionality.

A mock integration is used when there is no backend required. The developer of the API determines the specific response with the response generated directly from the API.

API Gateway Tools

Several AWS tools are available for creating and maintaining REST and WebSocket APIs . Tools include:

- a. AWS Management Console – Use the API Gateway service menu within the management console to configure APIs. This interface allows a user to visualize and configure each of the API Gateway contents and components.
- b. AWS CLI - A command-line interface you can use to leverage Amazon's API Gateway functionality.
- c. AWS SDK's – Python and other programming languages can be used to create API gateways programmatically.

Hello, API Gateway

This section provides a step through of creating a simple REST API using Amazon API Gateway. The processes to develop the API, Test the API within the AWS Management Console and deploy the API for subsequent testing using a Web URL are provided.

To develop a new REST API, login to your AWS Educate classroom and navigate to the API Gateway service. Select **"Create API"** to continue.

Next, we need to choose the protocol, select creation options, name and deccrip the API and select the Endpoint type. As shown in figure 1, select the **“REST”** protocol, **“New API”** for the Create New API option, name the API **“hello-1”**, provide a brief description and select a Regional as the endpoint type. We described this a mock hello since we won’t be using a backend service. The Regional endpoint will use the specified region (your default region) and intended to serve clients in the same AWS region to reduce latency. Select a New REST API:

The screenshot shows the 'Choose the protocol' section with 'REST' selected. Below it is the 'Create new API' section with 'New API' selected. The 'Settings' section shows the API name 'hello-1', description 'Mock hello', and endpoint type 'Regional'.

Choose the protocol

Select whether you would like to create a REST API or a WebSocket API.

☒ REST ☐ WebSocket

Create new API

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

☒ New API ☐ Clone from existing API ☐ Import from Swagger or Open API 3 ☐ Example API

Settings

Choose a friendly name and description for your API.

API name* hello-1

Description Mock hello

Endpoint Type Regional

Figure 1 Creating a New REST API

Scroll down and select **“create API”** to continue.

Next, we need to add a Method. To do so, select Actions arrow in the Resources and select **“GET method Create Method”**. A drop down box will appear the method type can be selected as shown in figure 2.

The screenshot shows the 'Resources' section with a blue bar containing a slash '/'. Below it is a dropdown menu with a checkmark icon and a close button. The 'Methods' section is visible on the right.

Resources Actions

/

Methods

Figure 2 Creating a Method

Use the arrow on the right side of the drop down box to select **“Get”**. Click the check box to continue and the select Mock as the Integration type as shown in figure 3.

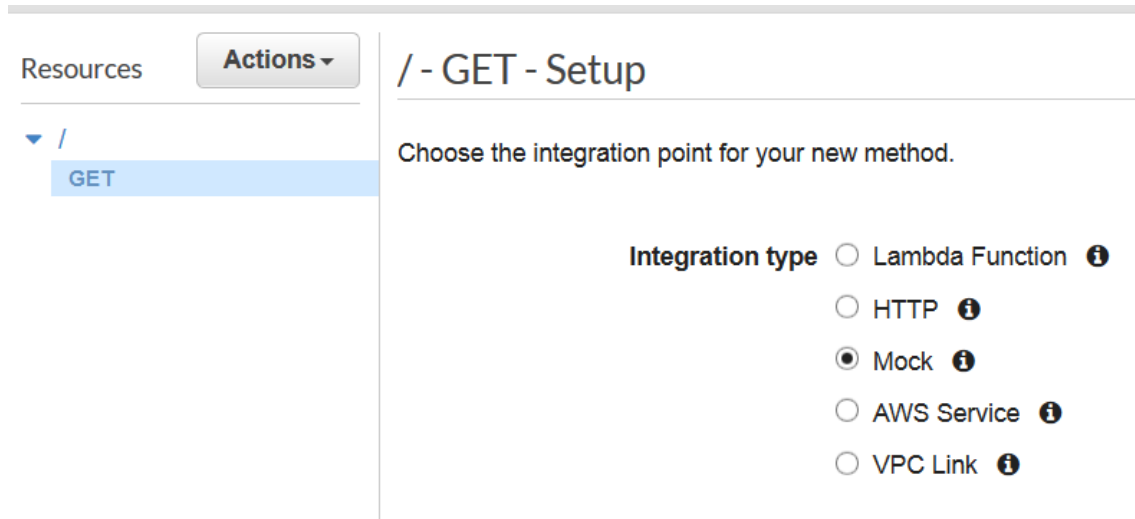


Figure 3 Selecting the Mock Integration Type

Click **“Save”** to continue.

This will result in the Method Request/Response and Integration Request/Responses to appear as shown in figure 4.

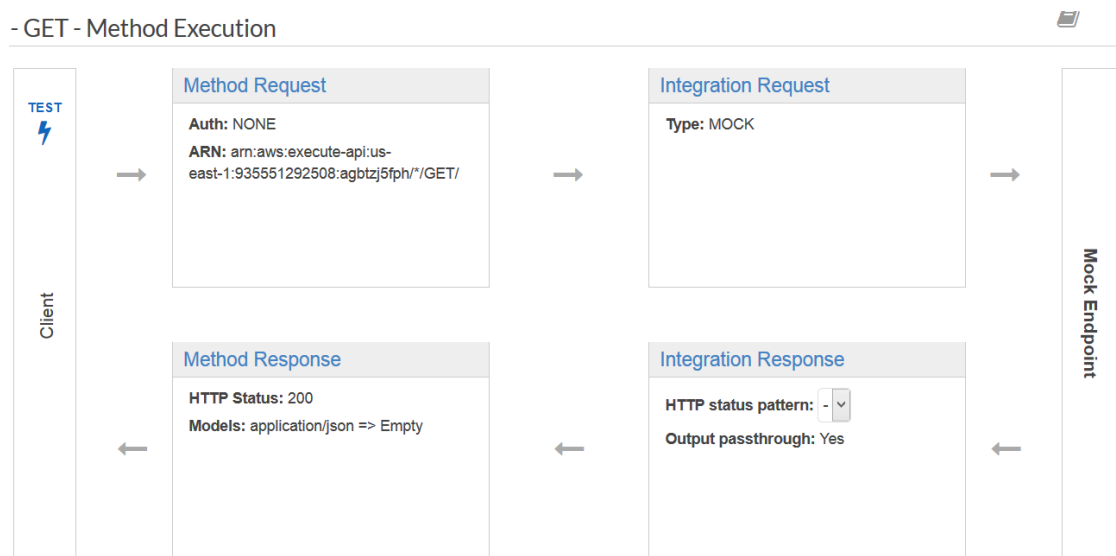


Figure 4 Method and Integration Request/Response

Since we used the Mock Integration type, we won't have backend component such as Lambda. We will integrate Lambda in future examples but the process is nearly identical except some functionality and backend service would need to Integrate in the execution path.

To properly test the API, we can use an input parameter in the form of a query string. We use query strings all of the time when we navigate Web pages or browse sites for products to purchase. The URL includes additional parameters which provide a parameter and an associated value. For example, the

following URL includes a query string with firstname and lastname parameter with values of John and Smith respectively:

```
https://www.example.com/test/data?firstname=John&lastname=Smith
```

As an additional example, consider the following parameter string that includes 5 parameters and associated values.

```
http://webapps.umuc.edu/soc/us.cfm?fAcad=UGRD&fAcad2=&fSess=2198&fLoc=ALL&fSubj=SDEV
```

The parameters always starts after the “?” character values assigned using the “=” assignment operator. Subsequent parameters are prefixed with a “&” character.



Knowing how query strings are formed is critical in both the design of the API and the eventual testing both within the AWS Management console and outside using a browser or other client.



To set-up a query string parameter click on the Method Request component of the Get Method Execution we just created. Becoming comfortable with navigating the API may take some time. To help with navigation be aware that the visualization includes multiple links inside the screen. For example, Clicking on “Method Execution” returns to the main diagram showing all 4 request and response types. Also, each request and response type is clickable. Hence, clicking the Method Request link will yield the view shown in figure 5.


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

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

Settings


Authorization NONE  

Request Validator NONE  

API Key Required false 

▼ URL Query String Parameters

Name	Required
No query strings	

 [Add query string](#)

▶ HTTP Request Headers

Figure 5 Selecting Method Request

Click on the URL Query String Parameters to continue.

To add the query string, click on the Add query string option and enter a parameter named “welcome”. Click the check icon to save the parameter. The results are shown in figure 6.

URL Query String Parameters

Name	Required	Caching	
welcome	<input type="checkbox"/>	<input type="checkbox"/>	 
 Add query string			

Figure 6 Adding the welcome parameter

A mapping template will be used to map the welcome query string parameter to an HTTP status code response. This essentially allows the API Gateway to reply to the request without having a specific backend service. To add a mapping template, click on the “**Method Execution**” link to return show all of the request and response options and then select “**Integration Request**”. Expand the Mapping Templates item. Click on the Request body passthrough option of “When there are no templates defined (recommended)” as shown in figure 7.

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

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


Integration type ☐ Lambda Function ⓘ
☐ HTTP ⓘ
☒ Mock ⓘ
☐ AWS Service ⓘ
☐ VPC Link ⓘ

Mapping Templates

Request body passthrough ☐ When no template matches the request Content-Type header ⓘ
☒ When there are no templates defined (recommended) ⓘ
☐ Never ⓘ

Content-Type

application/json




 [Add mapping template](#)

Figure 7 Integration Request body passthrough

Recapping what we have so far, we have set-up a GET request that will send a value with parameter named “welcome” to be processed. Since we don’t have backend to handle the request at this time, we need some logic that will return a status code based on the value of the parameter. For this simple example, we will pass a successful status code of 200 if the value of the welcome parameter is “Dr. Robertson”. For other values, we will return a status code of 500.

To enter this code logic, click on the application/json form element and enter the following code.

```
{
  #if( $input.params('welcome') == "Dr. Robertson" )
    "statusCode": 200
  #else
    "statusCode": 500
  #end
}
```

Figure 8 illustrates the process to enter the code logic for this API.

Mapping Templates

- Request body passthrough** ☐ When no template matches the request Content-Type header ⓘ
- ☒ When there are no templates defined (recommended) ⓘ
- ☐ Never ⓘ

Content-Type	
application/json	⊖

+ Add mapping template

application/json

Generate template:

```
1 {
2   #if( $input.params('welcome') == "Dr. Robertson" )
3     "statusCode": 200
4   #else
5     "statusCode": 500
6   #end
7 }
```

Figure 8 Entering Application Logic

Click Save to continue.

We aren’t finished yet. We still have align a specific response associated with the status codes. This action is performed in Integration Response view.

Navigate to the Integration Response view by selecting **“Method Execution”** followed by **“Integration Response”**. Select the 200 response, expand the Mapping template and provide the following json/application response.

```
{
  "statusCode": 200,
  "message": "Hello Dr. Robertson from API Gateway!"
}
```

Figure 9 shows the end results after entering the code. Be sure to click the **“Save”** associated with the Mapping Templates or you code will not be saved.

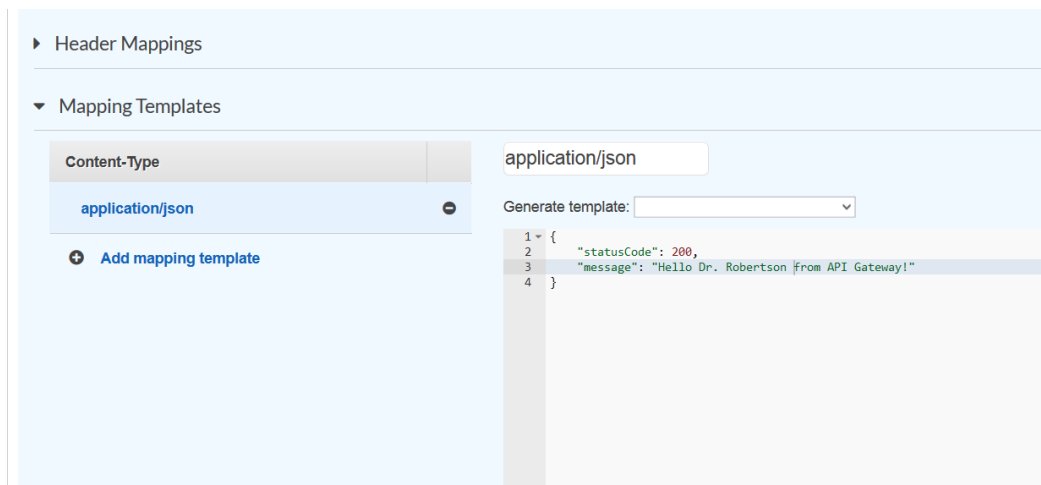


Figure 9 Saving the Integration Response for 200 Status Code

We now have a response for the status code of 200 but we still need a response when the status code of 500 is returned. To add this logic, click to display the Method Response and then select **“Add an Response.”** See figure 10.

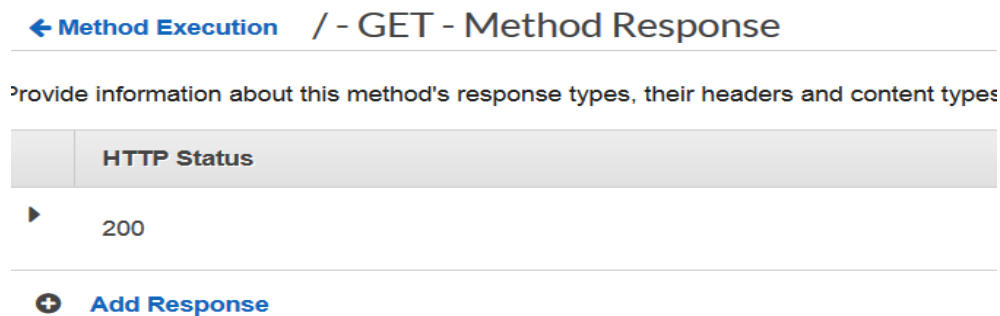


Figure 10 Adding a Method Response

Enter 500 in the HTTP textfield then save the value by selecting the checkmark icon as shown in figure 11.

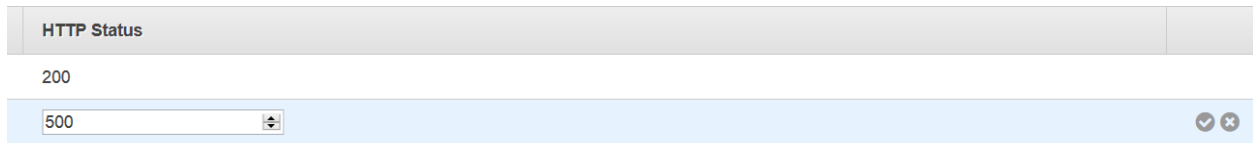


Figure 11 Adding the 500 HTTP Status Code

Now that we have added a status code method response, we need to create a mapping template similar to what we did for the status code of 200. To do this return the Method Execution -> Integration Response view as shown and choose Add Integration Response. We want any possible 500 level status code to be provide the same 500 level response message. To accomplish use a regular expression of `5\d{2}` (i.e. 5 and any other 2 digit patterns) and align with the 500 Method response status as shown in the figure 12.

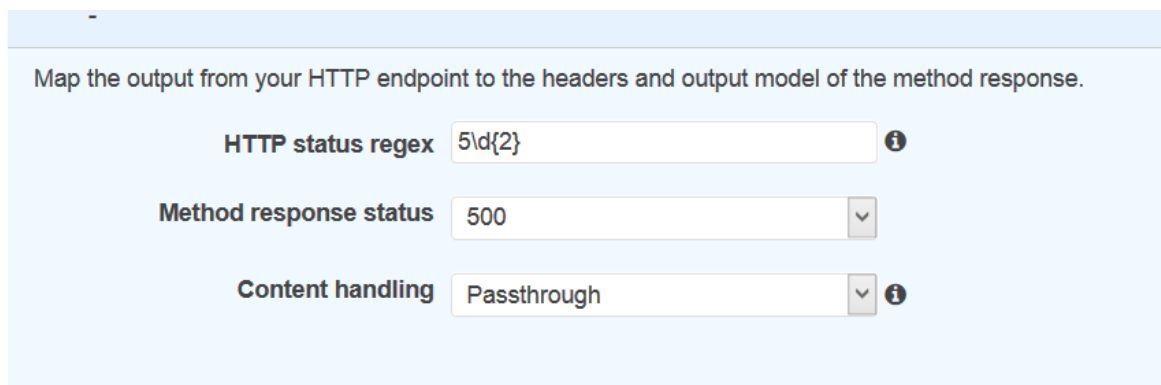


Figure 12 Setting up the Method Response Map

Similar to our 200 status code mapping template, expand the mapping template option, select application/json and add the following (or similar) code.

```
{
  "statusCode": 500,
  "message": "You didn't say the magic word."
}
```

Resulting in the following view shown in figure 13.

HTTP status regex: 5ld{2}

Content handling: Passthrough

Header Mappings

Mapping Templates

Content-Type	
application/json	-
Add mapping template	

Generate template:

```

1 {
2   "statusCode": 500,
3   "message": "You didn't say the magic word."
4 }

```

Figure 13 Entering the Integration Response for Status Code 500

Be sure to click the **“Save”** button to save your mapping template.

Finally, we can test the Mock Integration. Since we are using query strings to enter the parameter and values, we will need to input the correct test values.

Select the Method Execution link to return the main Get Method Execution view. Then select the TEST link on the left side of the screen as shown in figure 14.

' - GET - Method Execution

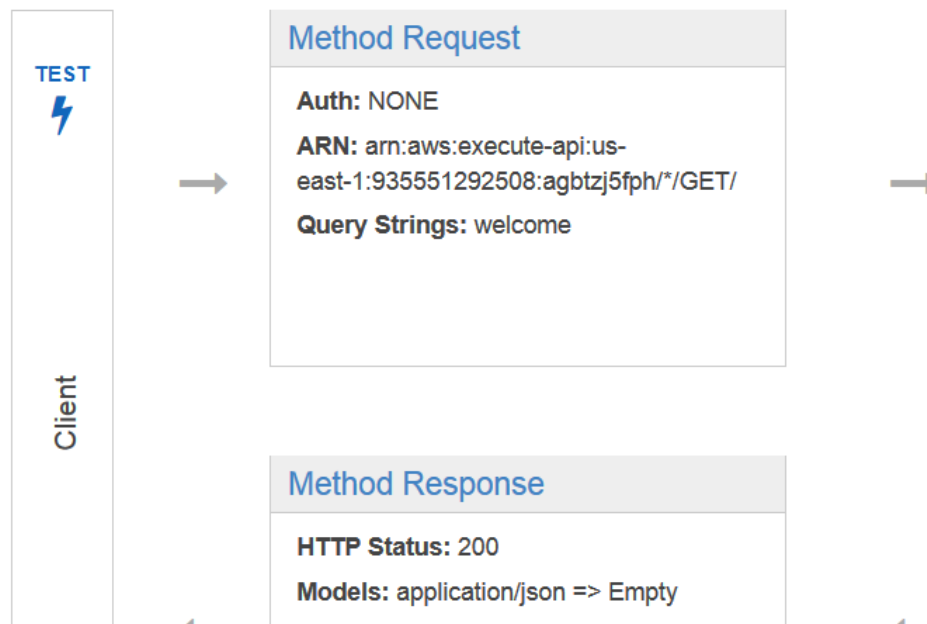


Figure 14 Testing the Mock Integration

Enter `welcome=Dr. Robertson` in the Query Strings textfield and then scroll down and select “Test”. The results are shown in figure 15. Notice in this case, the status code was 200 because the expected welcome value of “Dr. Robertson” was entered.

Make a test call to your method with the provided input

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

Query Strings
`welcome=Dr. Robertson`

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

Request: `/?welcome=Dr. Robertson`
Status: 200
Latency: 4 ms
Response Body

```
{
  "statusCode": 200,
  "message": "Hello Dr. Robertson from API Gateway!"
}
```

Response Headers

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

Figure 15 Status Code 200 Response

To test the 500 status code enter any other value (including null) in the Query Strings. For example:

`welcome=John`

As shown in figure 16, the status code of 500 results are a string other than the expected string was entered.

Make a test call to your method with the provided input

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

Query Strings
`welcome=John`

Headers

Request: `/?welcome=John`
Status: 500
Latency: 5 ms
Response Body

```
{
  "statusCode": 500,
  "message": "You didn't say the magic word."
}
```

Figure 16 Status Code 500 Response

Deploy the API

Now that some testing has been performed we can deploy for additional testing or into our production environment. To deploy, select the Actions drop down option then select Deploy API as shown in figure 17.

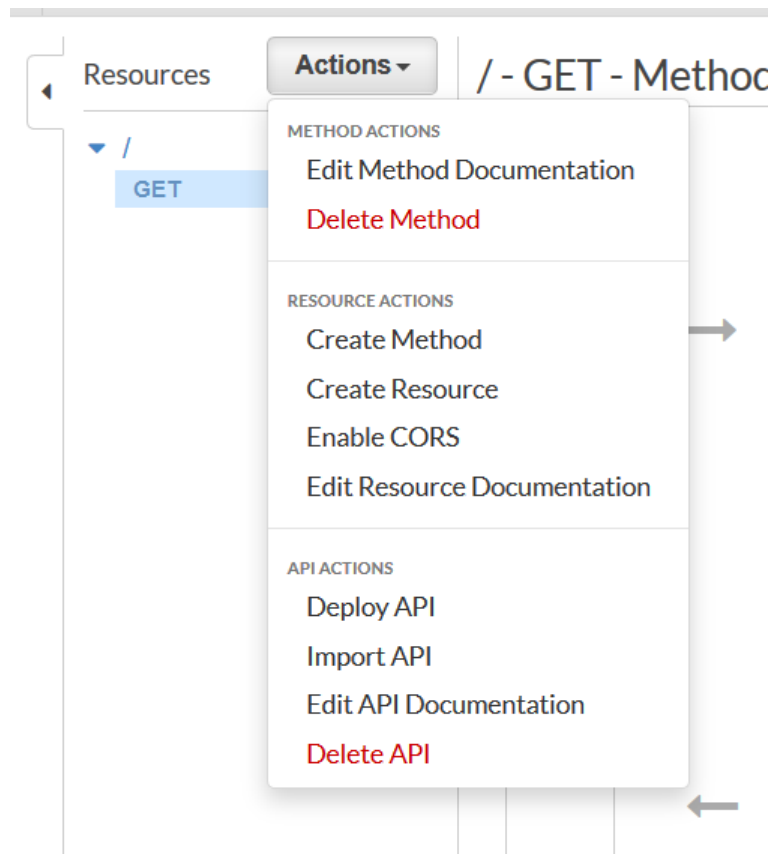


Figure 17 Deploy the API

When the deploy API screen appears select a new stage and enter an appropriate stage name, description and deployment description. In our example, we will call the stage “qa” representing a quality assurance stage as shown in figure 18.

Deploy API

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

Deployment stage

[New Stage]

Stage name*

qa

Stage description

Quality Assurance

Deployment description

Additional Testing

Cancel

Deploy

Figure 18 Creating the QA stage

Select **“Deploy”** to deploy the API.

Upon successful deployment a URL is provided with the format we previously discussed. For example:

<https://cjri6mqqwi.execute-api.us-east-1.amazonaws.com/qa>

Figure 19 shows an example screen after the QA Stage is deployed along with the results URL.



Figure 19 API is successfully deployed

Once deployed, multiples options for enabling cache, throttling and other options are available to experiment with. For the purposes of this class and exercise, we will just leave the default settings.

Now, you can test the URL using your default browser by typing the provided API into the URL window. Be sure to include the parameter and test value in the query string:

<https://cjri6mqqwi.execute-api.us-east-1.amazonaws.com/qa?welcome=DrRobertson>

Figure 20 shows the response when the expected welcome value was entered.



Figure 20 Expected Response for Status Code 200

Figure 21 shows the response when another string is entered.

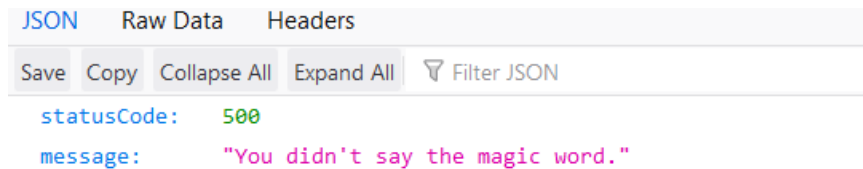


Figure 21 Status Code 500 Response

Clearly, there are many detailed steps to get to the final output for a fairly simple application. You should experiment with the process for several simple Mock Integrations to become comfortable with the steps and the purpose of each step. Additional more complex code can be created using the same process.

Next, we will discuss using a backend service (e.g. Lambda) as opposed to the Mock Integration.

API Backend Integration with Lambda

For this example, we will create a Lambda function that allows a user to input 3 numbers and return the sum. Then we will use this as backend service for an API gateway.

Creating the Lambda Function

First, let us create the Lambda function. You may need to revisit the details associated with creating a Lambda function in the previous week examples as this walk-through will just provide the high-level steps.

From the AWS management console, navigate to the Lambda service and create a new Lambda function. You should build the code from scratch as before. The following Python code will suffice for this simple application:

```
import json

def lambda_handler(event, context):
    # TODO implement
    number1 = event['Number1']
    number2 = event['Number2']
    number3 = event['Number3']
    sum = int(number1) + int(number2) + int(number3)
    return {
        'statusCode': 200,
        'body': json.dumps('Welcome to Addition Application!'),
        "Number1": number1,
        "Number2": number2,
        "Number3": number3,
        "Sum": sum
    }
```

Since Web HTTP parameters are always strings, the `int()` was added to convert to a number to properly interface with the API Gateway. Be sure to save your new code and create a test function with 3 numbers as input:

```
{
  "Number1": 1,
  "Number2": 4,
  "Number3": 5
}
```

Test your code to make sure it works before proceeding to the next step.

Figure 22 shows a screen capture of the newly created “SimpleAdd” Lambda function.

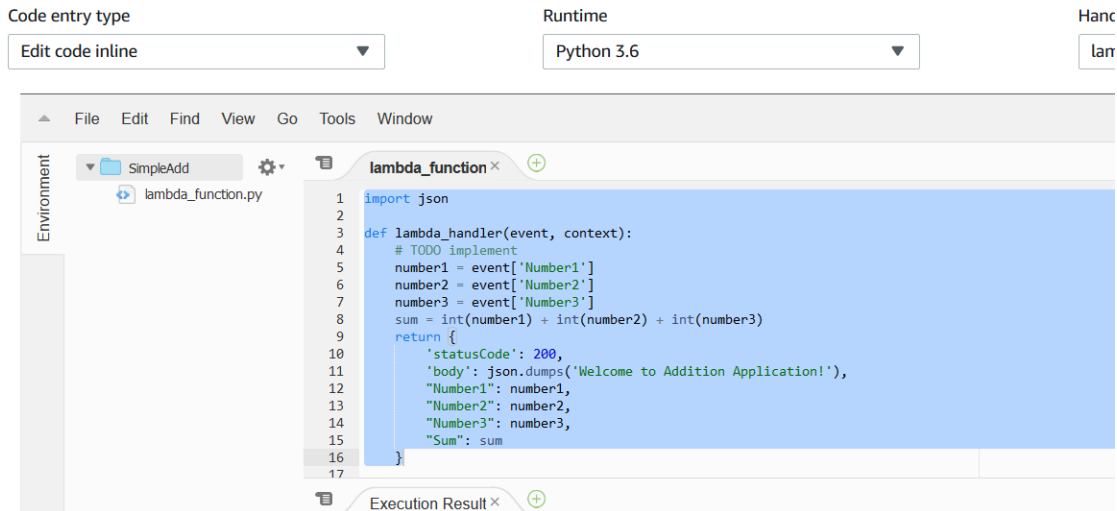


Figure 22 SimpleAdd Lambda Function

Figure 23 shows the output of testing the Lambda function verifying the function works as expected.

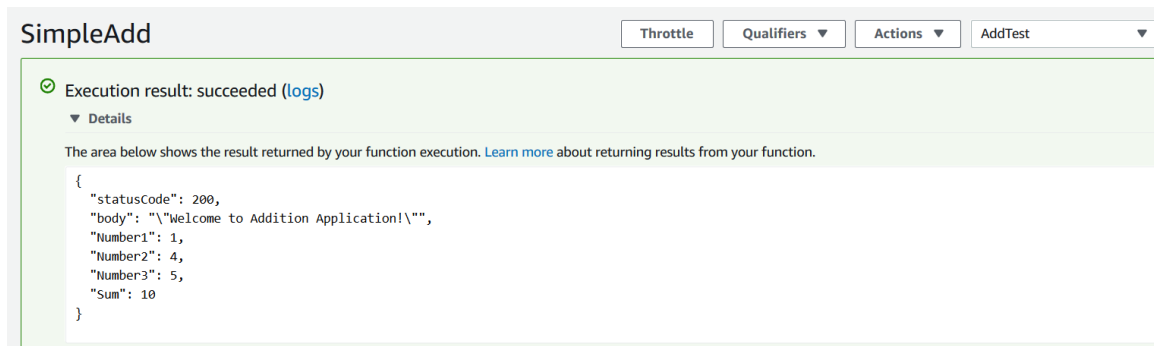


Figure 23 Lambda Function Successful Test

Creating the API Gateway

Now that the Lambda function has been created and tested, we can create an API Gateway that uses the Lambda function backend service.

To create a new API Gateway, navigate to the API Gateway service, select new API and select the REST protocol. In this exercise, we will name the API Gateway AddAPI.

As previously described, create a new GET method using the Action->Create Method option. This time instead of selecting Mock Integration we will use the Lambda function and use the Lambda function textfield to enter the name of the Lambda function we just created as shown in figure 24.

/ - GET - Setup

Choose the integration point for your new method.

Integration type ☒ Lambda Function ⓘ
☐ HTTP ⓘ
☐ Mock ⓘ
☐ AWS Service ⓘ
☐ VPC Link ⓘ

Use Lambda Proxy integration ☐ ⓘ

Lambda Region ▼

Lambda Function

Use Default Timeout ☒ ⓘ

Figure 24 Setting up Lambda Integration

Click **“Save”** to continue. You will most likely receive a warning message about Adding permission to the Lambda function. This is expected as without permissions the integration will not work. (See figure 25)

Add Permission to Lambda Function

You are about to give API Gateway permission to invoke your Lambda function:
arn:aws:lambda:us-east-1:935551292508:function:SimpleAdd

Figure 25 Adding Permissions

If you are monitoring your Lambda function, you will notice this permissions for API Gateway has now been added to your designer view. Figure 26 shows the results of navigating to the Lambda function and viewing the Designer view. Notice the addition of the API Gateway as a trigger.

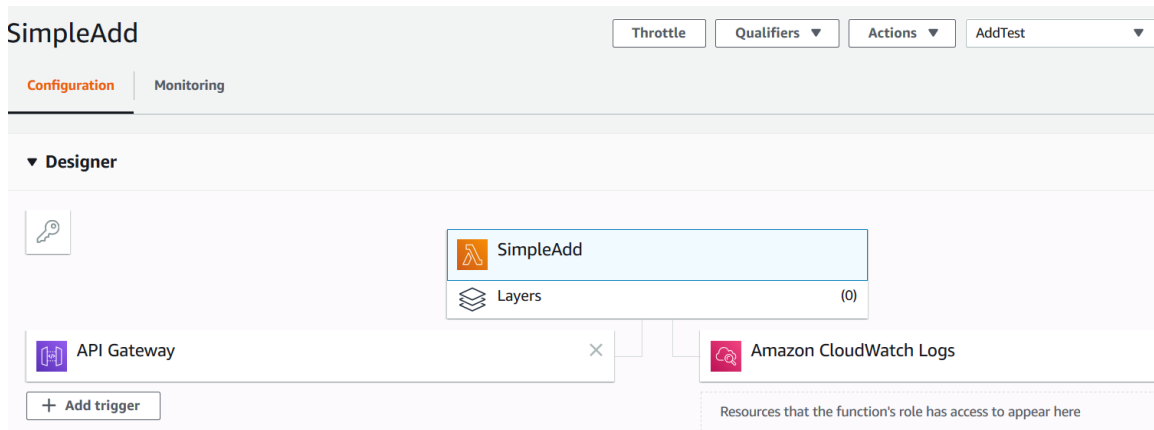


Figure 26 Confirming the API Gateway Trigger

Figure 27 shows the results of the Get-Method Execution for the API Gateway.

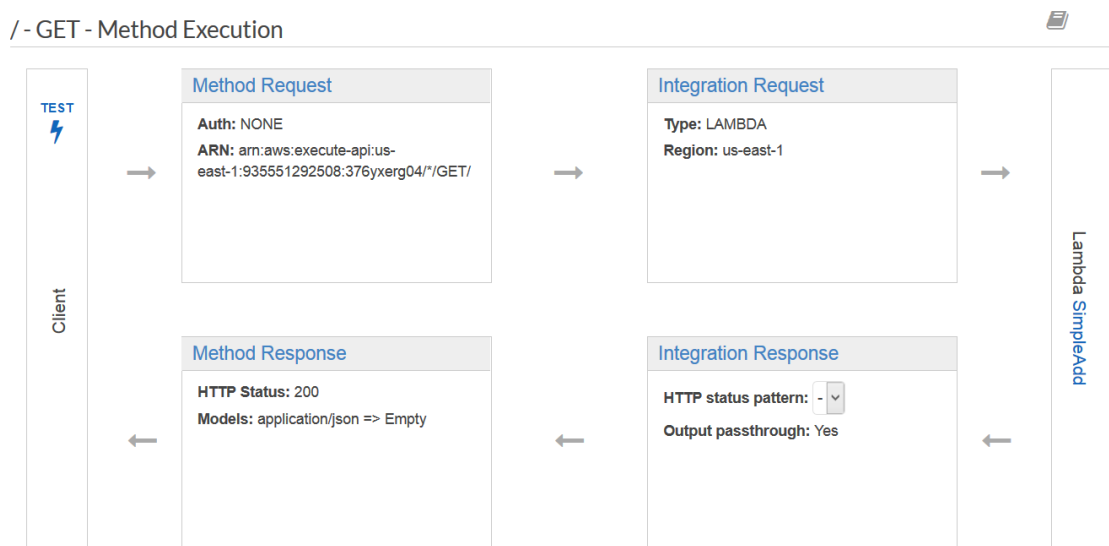


Figure 27 Method Execution

Since the API will taking input parameters we do need to configure the values using the mapping template in the Integration response. From the Get-Method Execution, select Integration Request and the expand the Mapping Templates to expose the options. As shown in figure 28, be sure to select “When there are no templates defined(recommended)” and use the content type of application/json. The following json syntax should be used to align with the 3 input Numbers.

```
{
  "Number1": "$input.params('Number1')",
  "Number2": "$input.params('Number2')",
  "Number3": "$input.params('Number3')",
}
```

▼ Mapping Templates

- Request body passthrough**
- ☐ When no template matches the request Content-Type header ⓘ
 - ☒ When there are no templates defined (recommended) ⓘ
 - ☐ Never ⓘ

Content-Type	
application/json	⊖

+ Add mapping template

application/json

Generate template:

```
1 {  
2   "Number1": "$input.params('Number1')",  
3   "Number2": "$input.params('Number2')",  
4   "Number3": "$input.params('Number3')"  
5 }
```

Figure 28 Adding Mapping Template

Scroll down and click “Save” to continue.

Finally, we can deploy the API to test the integration with the Lambda backend service. From the Actions menu select “Deploy API”.

As shown in figure 29, enter appropriate Stage descriptors and select **“Deploy”**.

Deploy API

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

Deployment stage

[New Stage]

Stage name*

vers1

Stage description

Lambda Integration

Deployment description

First test

Cancel
Deploy

Figure 29 Deploying the API

To test the API, take the returned URL and build the query string for different test cases. For example, the following should result in a sum of 8 since 3+4+1 would be 8.

<https://376yxerg04.execute-api.us-east-1.amazonaws.com/vers1?Number1=3&Number2=4&Number3=1>

Figure 30 shows the results of running this test case.

JSON	Raw Data	Headers
Save	Copy	Collapse All
	Expand All	Filter JSON
statusCode:	200	
body:	"Welcome to Addition Application!"	
Number1:	"3"	
Number2:	"4"	
Number3:	"1"	
Sum:	8	

Figure 30 API Results for summing 3 numbers

Be sure to experiment with creating REST API's for both Lambda and Mock Integrations. Your next homework will demonstrate your understanding of these topics.