

Build RESTful Microservices with AWS Lambda and API Gateway

Description

Amazon API Gateway allows you to design RESTful interfaces and connect them to your favorite backend. You can design your own resources structure, add dynamic routing parameters, and develop custom authorizations logic. Each API resource can be configured independently, while each stage can have specific cache, throttling, and logging configurations.

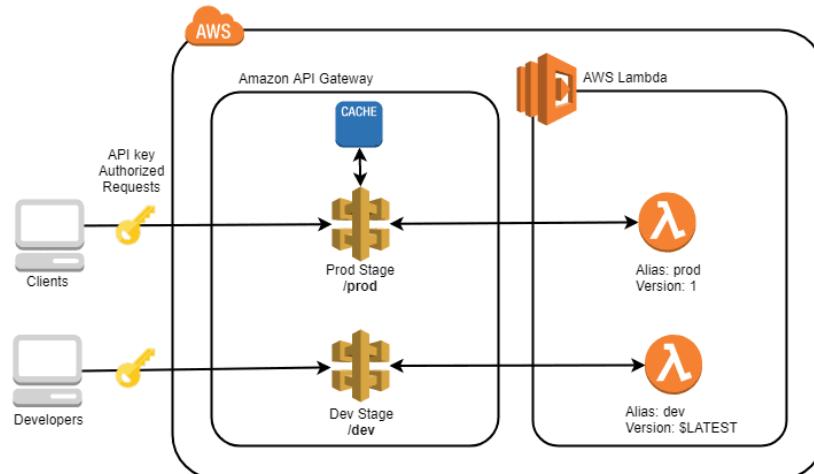
This approach is particularly useful when you consider that each request and response can be attached to a custom mapping template, in order to perform custom data manipulation or improve API backward compatibility.

In this Lab, you will see how to define a simple API and how to connect it to AWS Lambda. This provides a nice way to obtain a scalable backend for modern web applications or mobile apps. You will configure custom stages, protect resources with an API key, and explain how to best connect API Gateway stages with AWS Lambda versions and aliases. You will learn about AWS Lambda's basic configuration, monitoring, and versioning as you progress through the Lab.

Lab Objectives

Upon completion of this Lab, you will be able to:

- Understand the basics of RESTful APIs
- Implement REST APIs using Amazon API Gateway
- Enable desirable API features in API Gateway including caching, throttling, CORS, usage plans, and API key access
- Create serverless API backends using AWS Lambda functions
- Implement best practices for integrating Lambda backends in API Gateway



Lab steps

- Logging In to the Amazon Web Services Console
- Understanding RESTful APIs
- Defining New API Gateway Resources
- Creating an AWS Lambda Function Backend
- Versioning and Aliasing the Lambda Function
- Configuring the API Gateway Backend
- Following Best Practices for Versions, Aliases, and Stages
- Creating API Keys and Usage Plans
- Enabling CORS on API Gateway Resources
- Enabling API Gateway Caching and Throttling
- Cleaning up API Resources and Lambda Functions

Understanding RESTful APIs

Introduction

The RESTful approach makes the development of modern web applications much more flexible and maintainable.

The new tendency is to build more complex clients with client-side frameworks such as AngularJS, ReactJS, PolymerJS, etc. This way, your web app can easily be distributed as a set of static assets - HTML, JavaScript and CSS files - which will load dynamic content via API.

This new architectural pattern allows you to separate business logic from your presentation layer(s). At the same time, your services will be easier to scale and reuse, eventually by more than one client, including your own mobile apps too.

In the AWS world, a typical configuration looks similar to the following:

- Static website hosted on **Amazon S3** and distributed via **Amazon CloudFront** ([learn more, opens in a new tab](#)).
- RESTful API implemented with **AWS Lambda** and HTTP endpoints exposed via **Amazon API Gateway**.
- Dynamic data stored in DynamoDB, RDS or other database as a service alternative.

This is how you'd build a completely serverless web application, meaning that you won't need to manage, patch or maintain any server during your development and deployment workflow.

What is REST?

REST stands for *Representational state transfer* and is meant to be an architectural reference for developing modern and user-friendly web services.

Instead of defining custom methods and protocols such as SOAP or WSDL, **REST is based on HTTP** as the transport protocol. HTTP is used to exchange textual representations of web resources across different systems, using predefined methods such as *GET*, *POST*, *PUT*, *PATCH*, *DELETE*, etc.

The standard representation format is **JSON**, which is also the most convenient format to develop modern web applications since it's natively supported by JavaScript.

The level of abstraction provided by a RESTful API should guarantee a **uniform interface** and a set of **stateless interactions**: this means that all the information necessary to process a request must be included in the request itself (i.e. URL, headers, query string or body). Furthermore, each resource should be eventually cachable by the client, based on the particular use case.

What's next?

With **Amazon API Gateway**, you can define resources, map them to custom models, specify which methods are available (i.e. GET, POST, etc.) and eventually bind each method to a particular Lambda function. Alternatively, you can attach more than one method to one single Lambda function. This way, you will maintain fewer functions and partially avoid the cold-start Lambda issue.

In the next steps, you will learn how to create new resources/endpoints and how to implement a serverless backend for your API.

Defining New API Gateway Resources

Introduction

During this lab, you will design a very simple API to read a list of items and retrieve the details of a given item by ID.

Therefore, you will define two HTTP endpoints:

- `/items/`
- `/items/{ID}/`

These routes are compliant with the RESTful design principles: the first endpoint lets you retrieve the full list of items, while the second one corresponds to the detail of a single item given its ID. As you can notice, the second route is defined in terms of a dynamic parameter, which is part of the URL itself. Of course, you could achieve the very same result with a query string parameter (i.e. `/items/?ID=XXX`), but this pattern is generally discouraged.

In order to keep your data structures well-defined and your interface documented and maintainable, you can create new API Gateway Models.

Models are useful to map your API endpoints payload, via mapping templates. They are not strictly required, as you can manually define your mapping templates, but having a model will let you generate strongly-typed SDKs for your API resources and better validate your payloads.

Technically, you can define a model with a [JSON Schema](#). For this lab, you will define this simple model:

[Copy code](#)

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "Item Schema",  
  "type": "object",  
  "properties": {  
    "ID": {"type": "number"},  
    "name": {"type": "string"},  
    "price": {"type": "number"}  
  }  
}
```

Our items will have a unique numeric ID, a string identifying its name, and a numeric price. This could be an overly simplified model for a sample e-commerce application.

In this lab step, you will:

1. Create a new API in Amazon API Gateway and define the model explained above
2. Create associated API resources and methods
3. Deploy the API

Instructions

1. In the AWS Management Console search bar, enter *API Gateway*, and click the **API Gateway** result under **Services**
2. Click **Create an API**, then click **Build** in the **REST API** card

REST API

Develop a REST API where you gain complete control over the request and response along with API management capabilities.

Works with the following:
Lambda, HTTP, AWS Services

Import

Build

3. In the **Create new API** form, select **New API** and enter the following values:

- **API name:** *ItemsAPI*
- **Description:** *Cloud Academy Lab API*
- **Endpoint Type:** **Regional**

API details

New API
Create a new REST API.

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

Import API
Import an API from an OpenAPI definition.

Example API
Learn about API Gateway with an example API.

API name
ItemsAPI

Description - optional
Cloud Academy Lab API

API endpoint type
Regional APIs are deployed in the current AWS Region. Edge-optimized APIs route requests to the nearest CloudFront Point of Presence. Private APIs are only accessible from VPCs.
 Regional

IP address type | [Info](#)
Select the type of IP addresses that can invoke the default endpoint for your API.

IPv4
Supports only edge-optimized and Regional API endpoint types.

Dualstack
Supports all API endpoint types.

[Cancel](#) [Create API](#)

4. Click **Create API**.

The API is now created and you can define a model for it.

5. Click **API > ItemsAPI > Models** in the left navigation panel and click **Create model** to define a new API Gateway model:

Models (2)

[Delete](#)

[Edit](#)

[Update documentation](#)

[Create model](#)

Use models to define the format for the body of different requests and responses used by your API.

Name	Content type	Description
Empty	application/json	This is a default empty schema model
Error	application/json	This is a default error schema model

6. In the **New Model** form, enter the following values and then click **Create**:

- **Model name:** *Item*
- **Content type:** *application/json*
- **Model schema:**

Copy code

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "Item Schema",  
  "type": "object",  
  "properties": {  
    "ID": { "type": "number" },  
    "name": { "type": "string" },  
    "price": { "type": "number" }  
  }  
}
```

Name	<input type="text" value="Item"/>
Content type	<input type="text" value="application/json"/>
Description - optional	<input type="text"/>
Model schema	<pre>1 { 2 "\$schema": "http://json-schema.org/draft-04/schema#", 3 "title": "Item Schema", 4 "type": "object", 5 "properties": { 6 "ID": { "type": "number" }, 7 "name": { "type": "string" }, 8 "price": { "type": "number" } 9 } 10 }</pre>

The new Item model is listed under **Models** in the left navigation panel.

7. Create another model for a list of items with the following values:

- **Model name:** *Items*
- **Content type:** *application/json*
- **Model schema:**

[Copy code](#)

```
{
  "id": "Items",
  "title": "Items",
  "type": "array",
  "items": {"$ref": "https://apigateway.amazonaws.com/restapis/YOUR_API_ID/models/Item"}
}
```

Note: You need to replace YOUR_API_ID in the items JSON entry above with your actual API ID. You can find your API ID in the breadcrumb trail of the API Gateway Console (**Amazon API Gateway > APIs > ItemsAPI (*YOUR_API_ID*)**):



[API Gateway](#) > [APIs](#) > [Resources - ItemsAPI \(cd8fqlrx74\)](#)

You should now see both **Item** and **Items** beneath Models. You are now ready to create API resources and methods.

8. Click on **Resources** in the left navigation pane, then **Create Resource**.

Resources are the key abstraction for information in a RESTful API. You will create resources for lists of items and individual items.

9. In the **New Child Resource** form, enter the following values and then click **Create Resource**:

- **Proxy resource:** unchecked
- **Resource Path:** /
- **Resource Name:** *items*
- **Enable API Gateway CORS:** unchecked

Resource details

Proxy resource [Info](#)

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

Resource path

/

Resource name

items

CORS (Cross Origin Resource Sharing) [Info](#)

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

[Cancel](#)

[Create resource](#)

The **/items** resource is now selected in the **Resources** panel. Notice there are no methods created yet. You will add a GET method to retrieve a list of items.

10. Click on **Create Method** and enter the following values and then click **Create method**:

- **Method type:** GET
- **Integration type:** Mock

Method type

GET

Integration type

Lambda function
Integrate your API with a Lambda function.



HTTP
Integrate with an existing HTTP endpoint.



Mock
Generate a response based on API Gateway mappings and transformations.



The mock integration allows you to define API resources and methods whenever you don't have a real backend implementation yet. The advantage of this is that you can now use the **Items** model previously created to generate fake data. For example, this would allow you to provide a temporary API endpoint to your frontend/mobile developer, even before you have a real implementation.

11. Click on **Integration Response** tab

12. In the **Default - Response** section, click **Edit**, then expand the **Mapping templates** section:

Default - Response

[Edit](#)**HTTP status regex** [Info](#)

-

Content handling [Learn more](#)

Passthrough

Method response status code

200

Default mapping

True

13. Select **Items** from the **Generate template** drop-down:

▼ Mapping templates

Content type

application/json

Generate template

Items

**Template body**

```
1 ▪ {  
2   "id": "Items",  
3   "title": "Items",  
4   "type": "array",  
5   "items": {"$ref": "https://apigateway.amazonaws.com/restapis/kbftrsev9h/models/Item"}  
6 }
```



14. Overwrite the generated template with the following sample code to generate a list of 3 Item objects, with different IDs, prices and names:

[Copy code](#)

[

#foreach(\$ID in [1, 2, 3])

#set(\$price = \$ID * 1.5)

#set(\$name = "foo" + \$ID)

{

"ID" : \$ID,

"name" : "\$name",

"price" : \$price

}

#if(\$foreach.hasNext),#end

#end

]

15. Click **Save**.

You will be redirected to the **Resources** page.

16. From the **Resources** page, select the **Test** tab

17. Click the **Test** button.

The response should resemble the following:

Test

/items - GET method test results		
Request	Latency ms	Status
/items	4	200
Response body	[{ "ID" : 1, "name" : "foo1", "price" : 1.5 } , { "ID" : 2, "name" : "foo2", "price" : 3.0 } , { "ID" : 3, "name" : "foo3", "price" : 4.5 }]	

The method returns the template you defined. You will now define an API resource for individual items with the /items/{ID} path.

18. Scroll to the top of the **Resources** page and click **Create resource**.

19. In the **CreateResource** form, enter the following values and then click **Create resource**:

- **Proxy resource**: unchecked
- **Resource Path**: /items/
- **Resource Name**: {ID}
- **CORS (Cross Origin Resource Sharing)**: unchecked

Resource details

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

Resource path <input type="text" value="/items"/> <input type="checkbox"/> CORS (Cross Origin Resource Sharing) Info Create an OPTIONS method that allows all origins, all methods, and several common headers.	Resource name <input type="text" value="{ID}"/>
--	---

[Cancel](#) [Create resource](#)

You only need to insert the {ID} part of the path since **/items/** is automatically added when creating a new resource with **/items** selected in the resource panel. This means the Item resource is a child of the Items resource. The {ID} path variable makes the path dynamic. This way, API Gateway will treat any value as a dynamic ID and pass it to the backend implementation, which will return the corresponding data.

20. Create a **GET** method with a **Mock** integration

Note: Refer to steps **10** and **11** if needed to create the **GET** method with a **Mock** integration.

21. Click **Integration Response** and modify **Mapping Templates** of the 200 response status to the following:

[Copy code](#)

```
#set($Integer = 1)

#set($ID = $Integer.parseInt($input.params("ID")))

#set($name = "foo" + $ID)

#set($price = $ID * 1.5)

{
    "ID" : $ID,
    "name" : "$name",
    "price" : $price
}
```

This template extracts the dynamic ID parameter, casts it to an integer, and returns an Item object with fake generated data.

Note: Refer to steps **12 - 15** for modifying the **Mapping templates**.

22. Test the new **/items/{ID}** resource as you've just created. This time you need to enter a value under **ID** before clicking the **Test** button:

The screenshot shows the API testing interface with the following configuration:

- ID:** 1
- Query strings:** param1=value1¶m2=value2
- Headers:** header1:value1
header2:value2
- Client certificate:** No client certificates have been generated.

Below the configuration, the **Test** button is highlighted. The test results are displayed in a table:

① /items/{ID} - GET method test results		
Request	Latency ms	Status
/items/1	4	200
Response body	{ "ID" : 1, "name" : "fool", "price" : 1.5 }	
Response headers	{ "Content-Type": "application/json" }	

If you leave the ID field blank, an "Internal Server Error" is returned. Any integer value should return a valid JSON response. You will replace the simple Mock implementation with a real implementation in the following Lab Steps. For now, you can proceed to deploy the API with the mock backends.

Note: Refer to steps **16** and **17** for testing the resource.

23. Click **Deploy API** from the top of the page and select ***New Stage*** from the **Stage** drop-down menu. Enter the following values and then click **Deploy**:

Stage

New stage

Stage name

dev

i A new stage will be created with the default settings. Edit your stage settings on the **Stage** page.

Deployment description

Only mock items list

Cancel

Deploy

- **Stage name:** dev
- **Deployment description:** Only mock items list

The API is now deployed and ready to serve requests at the /items and /items/{ID} endpoints in the dev stage/environment.

The screenshot shows the AWS API Gateway Stages page for the 'ItemsAPI' API. On the left, there's a navigation sidebar with options like Resources, Stages, Authorizers, etc. The main area is titled 'Stages' and shows a single stage named 'dev'. A green success message at the top states: 'Successfully created deployment for ItemsAPI. This deployment is active for dev.' Below this, the 'Stage details' section shows the stage name 'dev', rate limit of 10000, burst limit of 5000, and both cache cluster and default method-level caching set to inactive. The 'Invoke URL' is listed as <https://cd8fqrlx74.execute-api.us-west-2.amazonaws.com/dev>. At the bottom, it says 'Active deployment' by 'lkorh4' on November 16, 2025, 20:20 (UTC+03:00). There are 'Edit' and 'Create stage' buttons at the top right of the stage card.

Note: You may see a warning notification that your user does not have **ListWebACLs** and **AssociateWebACL** permissions for Web Application Firewall (WAF Regional). Use of a Web Application Firewall is not required for this lab. You may see this warning several times during the lab. The warning can be safely ignored.

24. Repeat the **Deploy API** action from the **Resources** panel action again, this time creating the *prod* (production) stage.

Summary

In this lab step, you have deployed an API Gateway API in dev and prod stages by:

1. Creating a model
2. Creating API resources
3. Creating API resource methods using a mock backend integration
4. Deploying the API to each stage

In the next lab steps, you will create and implement a Lambda function as a backend of the two HTTP endpoints you have defined.

Creating an AWS Lambda Function Backend

Introduction

You designed and created an API Gateway resource in the previous Lab Step, but it doesn't have a real implementation yet.

In this lab step, you will create a new Lambda function that will handle both endpoints. This approach has the advantage of reducing the number of functions you need to maintain (and therefore the amount of code), besides partially solving the cold-start issue of AWS Lambda. The cold-start issue with Lambda is due to the initial startup phase of your functions code, which might take up to a few seconds (worst case). By having fewer functions, you increase the likelihood of keeping them warm even if you don't have a very high load.

The Lambda function you will create will dynamically check whether an Item ID or the entire list has been requested and behave accordingly. Since you already created an API, you will not use the simplified flow to create new Lambda-backed API Gateway resources. This flow is recommended whenever you start from scratch with Lambda and you don't need much in the way of API Gateway configuration, which is the norm in most simple use cases.

Instructions

1. In the AWS Management Console search bar, enter *Lambda*, and click the **Lambda** result under **Services**

You will see the **Functions** list page.

2. Click **Create function**:

3. In the **Author from scratch** form, enter the following values:

Function name

Enter a name that describes the purpose of your function.

ItemsFunction

Function name must be 1 to 64 characters, must be unique to

Runtime | Info

Choose the language to use to write your function. Note that

Python 3.13

- **Name:** *ItemsFunction*
- **Runtime:** Python 3.x (The latest version of Python 3)

Warning: The function name needs to be named *ItemsFunction*, otherwise you won't be able to proceed. Ensure that the function name matches *ItemsFunction* exactly, including capital letters.

4. To assign an IAM role, expand **Change default execution role**, and select the following:

- **Execution role:** Select **Use an existing role**
- **Existing role:** Select **lambda_basic_execution**

Execution role

Choose a role that defines the permissions of your function. To create

- Create a new role with basic Lambda permissions
 Use an existing role
 Create a new role from AWS policy templates

Existing role

Choose an existing role that you've created to be used with this Lambda function.

lambda_basic_execution

You have selected a role that has been created for you. The role has the required IAM permissions needed to complete this lab.

5. Scroll to the bottom and click **Create function**

6. Scroll down to the **Code source** section and overwrite the contents of the **lambda_function.py** file with the following code:

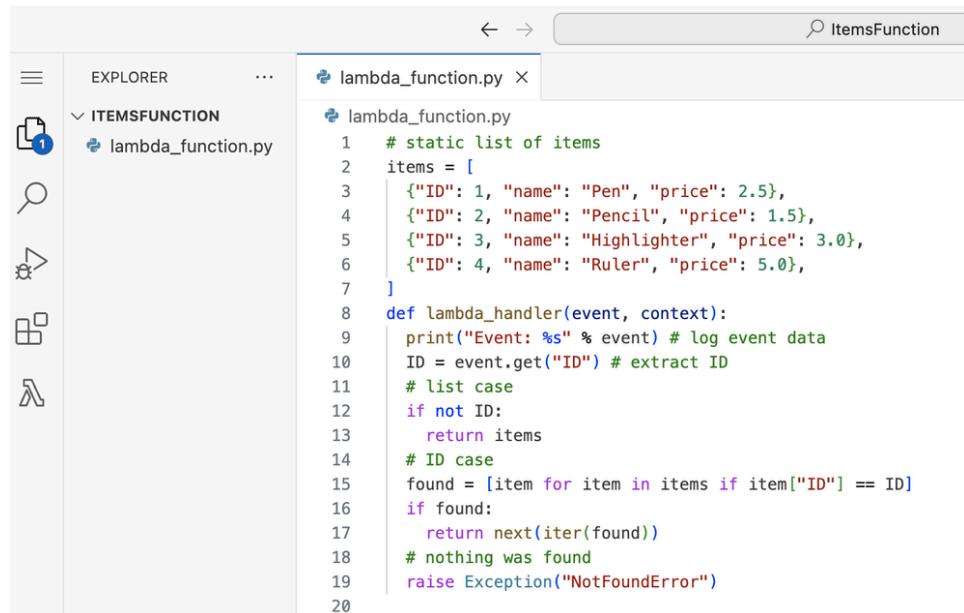
Copy code

```
# static list of items
```

```
items = [  
    {"ID": 1, "name": "Pen", "price": 2.5},  
    {"ID": 2, "name": "Pencil", "price": 1.5},  
    {"ID": 3, "name": "Highlighter", "price": 3.0},  
    {"ID": 4, "name": "Ruler", "price": 5.0},  
]  
  
def lambda_handler(event, context):  
    print("Event: %s" % event) # log event data  
    ID = event.get("ID") # extract ID  
  
    # list case  
    if not ID:  
        return items  
  
    # ID case  
    found = [item for item in items if item["ID"] == ID]  
    if found:  
        return next(iter(found))  
  
    # nothing was found  
    raise Exception("NotFoundError")
```

Your **Code source** section will look like this:

Code source [Info](#)



The screenshot shows the AWS Lambda code editor interface. On the left is the Explorer sidebar with a folder icon labeled 'ITEMSFUNCTION' containing 'lambda_function.py'. The main area is titled 'lambda_function.py' and contains the following Python code:

```
1 # static list of items
2 items = [
3     {"ID": 1, "name": "Pen", "price": 2.5},
4     {"ID": 2, "name": "Pencil", "price": 1.5},
5     {"ID": 3, "name": "Highlighter", "price": 3.0},
6     {"ID": 4, "name": "Ruler", "price": 5.0},
7 ]
8 def lambda_handler(event, context):
9     print("Event: %s" % event) # log event data
10    ID = event.get("ID") # extract ID
11    # list case
12    if not ID:
13        return items
14    # ID case
15    found = [item for item in items if item["ID"] == ID]
16    if found:
17        return next(iter(found))
18    # nothing was found
19    raise Exception("NotFoundError")
20
```

The function is very simple, without any specific dependency or infrastructure requirement.

The logic is straightforward:

- If no ID is given, return the whole list
- If the ID exists, return the corresponding Item
- Otherwise, raise an error

The other AWS Lambda configuration fields can be left at their defaults.

7. To deploy your function, click **Deploy** at the bottom left of the code editor

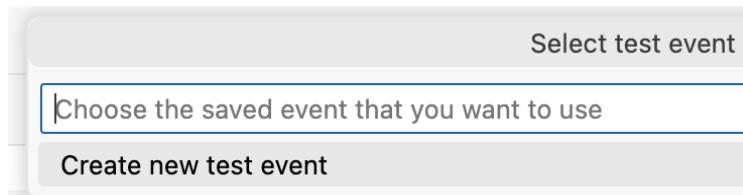
You will see a notification that your function has been deployed:

 Successfully updated the function **ItemsFunction**.

You can now test the function.

8. Below **Deploy** button click **Test**

In the pop-up, click **Create new test event**:



9. In the **Create new test event** form, enter the following values into the form:

- **Event name:** *GetItems*

- **Event body:** Replace the existing contents with the following JSON in the event editor text-area:
- Copy code
- 

Event Name
GetItems
Maximum of 25 characters consisting of letters, numbers, dots,

Event sharing settings
 Private
 This event is only available in the Lambda Console and to the event creator.
[Learn more](#)
 Shareable
 This event is available to IAM users within the same account who have permission to invoke shareable events. [Learn more](#)

Template - optional
Hello World

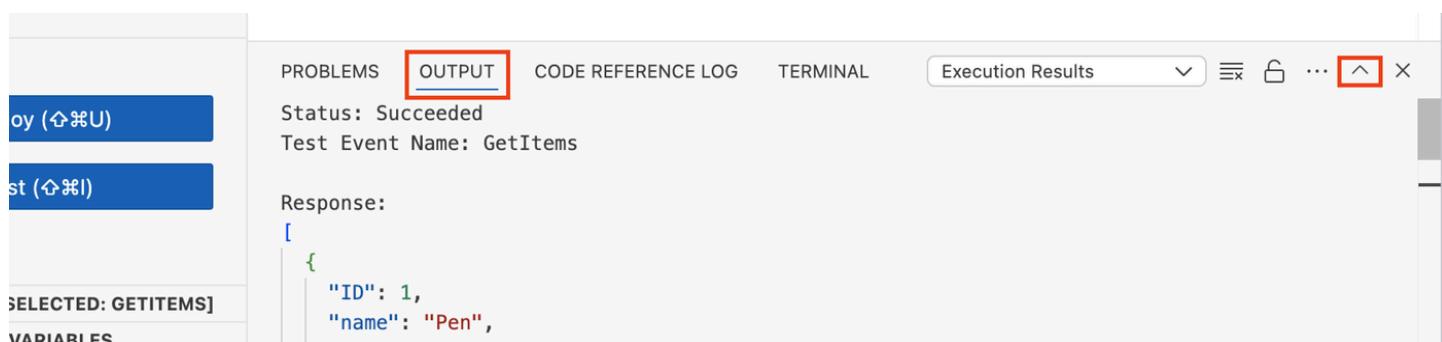
Event JSON
1 {}

The {} represents an empty object in JSON. Because no ID is specified the test will confirm the code path for returning all the items.

10. Scroll to the bottom and click **Save**.

11. Click **Test** once more to run the **GetItems** test you just created

Find and expand the **OUTPUT** tab. This tab contains the results of the test:



```

PROBLEMS OUTPUT CODE REFERENCE LOG TERMINAL Execution Results
Status: Succeeded
Test Event Name: GetItems

Response:
[
  {
    "ID": 1,
    "name": "Pen",
    ...
  }
]

```

The result confirms the list of all items is returned.

PROBLEMS OUTPUT CODE REFERENCE LOG TERMINAL

Status: Succeeded
Test Event Name: GetItems

Response:

```
[
  {
    "ID": 1,
    "name": "Pen",
    "price": 2.5
  },
  {
    "ID": 2,
    "name": "Pencil",
    "price": 1.5
  },
  {
    "ID": 3,
    "name": "Highlighter",
    "price": 3
  },
  {
    "ID": 4,
    "name": "Ruler",
    "price": 5
  }
]
```

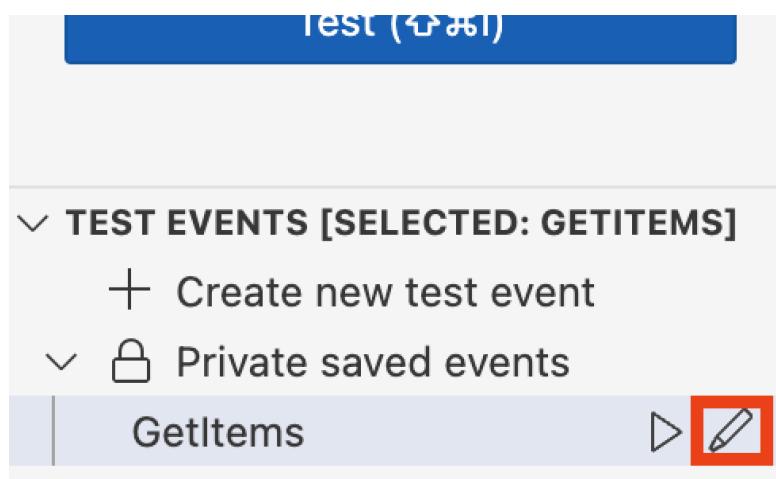
Function Logs:

```
START RequestId: 57a96539-f12c-4178-bbd6-824b63a2064a Version: $LATEST
Event: {}
END RequestId: 57a96539-f12c-4178-bbd6-824b63a2064a
REPORT RequestId: 57a96539-f12c-4178-bbd6-824b63a2064a Duration: 1.37
```

12. Repeat the test process to test two more code paths for the following cases:

- An object such as `{"ID": 1}`, which will return only one item.
- An object such as `{"ID": 5}`, which will return an `NotFoundError`.

You can find edit button within **TEST EVENTS** tab



Please note that in a real-world scenario you'd normally perform these operations on a database (DynamoDB, RDS, Firebase, etc.). In the simple scenario of the Lab, a static list of objects in main memory is more than enough. Also, keep in mind that the code will run on multiple machines (i.e. containers) and we will not be able to update the in-memory list consistently. In fact, only GET methods are defined and no transformation on the Items are allowed.

Summary

In this lab step, you created and tested the Lambda function that will serve as the backend for your API in API Gateway. Before you can use it, it is best to understand versioning so you can use different versions of the function in the dev and stage environments. The next lab step will show you how to work with function versions.

Versioning and Aliasing the Lambda Function

Introduction

In this lab step, you will create different versions of the AWS Lambda function corresponding with the dev and prod stages you created in API Gateway.

Each Lambda function has a default \$LATEST version, which is the one you can always work on and edit. Once your code is stable enough or whenever your code changes significantly, you can create a new *version*. A simple incremental number will be assigned to the new version and you will be able to use and test any version. As you can imagine, this is a useful mechanism to keep track of your functions history and eventually rollback to previous versions.

Using versions is helpful, but sometimes even that is not flexible enough. For example, you can only bind your API Gateway resources and methods to a specific version of your Lambda function: whenever you create a new Lambda Version, you'd need to update your API Gateway configuration as well. Fortunately, AWS provides the concept of *alias* to improve this situation.

An alias is a useful abstraction that allows you to refer to a function version without actually using a version number. For example, you may want to create a prod alias and connect it to your API Gateway production stage. You could do the same with a dev alias, bound to your development stage.

The best practice would be always referencing to aliases when configuring your API Gateway backend integrations. By default, API Gateway will point to the \$LATEST version, but you can always configure it to use a specific version or alias.

The following are examples of how the version and alias mappings work:

- **ItemsFunction** -> \$LATEST Version
- **ItemsFunction:1** -> Version 1
- **ItemsFunction:prod**-> prod alias -> Version 1

You will now create a new version and two new aliases for the Lambda function.

Instructions

1. In the AWS Lambda console, within **ItemsFunction**, click the **Versions** tab, then click **Publish new version**:

Aliases

Versions



Delete

Publish new version

A form will open allowing you to optionally specify a description of the new version.

2. To publish a new version, click **Publish**

Please note that you cannot modify published versions. You can only work on the **\$LATEST** version.

Next, you will create an alias for your AWS Lambda function.

3. At the top-right, click **Actions**, click the **Create alias** menu item, and enter the following form values:

- **Name:** dev
- **Description:** Development alias
- **Version:** \$LATEST

Name	dev
Description - optional	Development alias
Version	\$LATEST

You are creating an alias for the dev stage.

The dev stage will always work with the latest version of the function. By binding the dev alias to our **\$LATEST** version, you will be able to quickly implement changes and test them without explicitly publishing new versions. Of course, you can always adapt this configuration to your own needs. You usually don't want to be so aggressive in automatically releasing the latest function to production. You will alias prod to a specific version instead.

4. To save your alias and return to the function's details page, click **Save**, and then at the top click **ItemsFunction**.

5. To create an alias for production, at the top click **Actions, Create alias** and enter the following form values:

- **Name:** *prod*
- **Description:** *Production alias*
- **Version:** Select the number (not \$LATEST)

Name	prod
Description - optional	Production alias
Version	11

► **Weighted alias**

Notice the **Weighted alias** option. Traffic to an AWS Lambda function can be split between multiple versions. This allows you to test a new version of a function on a subset of traffic before fully deploying it. Weighted aliases aren't used in this lab.

6. To save your alias and return to the function's details page, click **Save**, and then at the top click **ItemsFunction**.

7. To see the aliases you created, click the **Aliases** tab:

The screenshot shows the AWS Lambda function details page. At the top, there are tabs for Code, Test, Monitor, Configuration, Aliases (which is underlined in blue), and Versions. Below the tabs, the Aliases section has a heading "Aliases (2) [Info](#)". There is a search bar with placeholder text "Filter by attributes or search by keyword". A table lists the aliases:

	Name	Versions
<input type="radio"/>	dev	version: \$LATEST (weight=100%)
<input type="radio"/>	prod	version: 11 (weight=100%)

You will see the **dev** and **prod** aliases listed.

You can also see the version you published by clicking on the **Versions** tab.

Summary

Now that the Lambda function is configured with versions and aliases, you can use it as an API Gateway backend. In the next step, you will properly configure API Gateway resources to use the Lambda function.

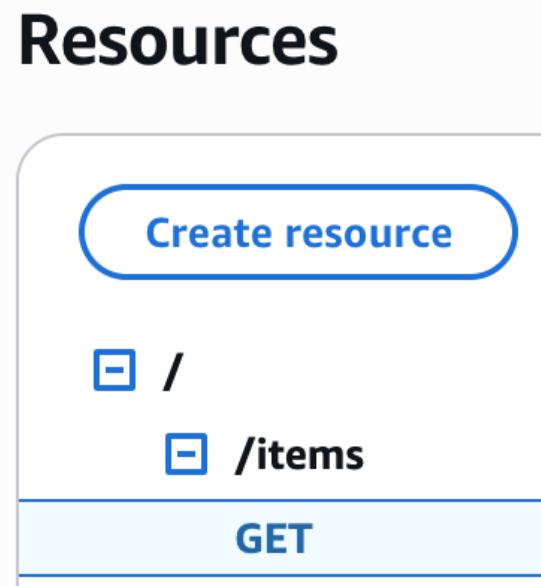
Configuring the API Gateway Backend

Introduction

In this lab step, you will update the API Gateway Integration Request from Mock to AWS Lambda. You will map both API endpoints to the same Lambda function.

Instructions

1. Return to the **API Gateway** console and select **/items > GET** method:



2. Click **Integration request > Edit** and set the following values:

- **Integration type: Lambda Function**
- **Use Lambda Proxy integration:** unchecked
- **Lambda Region:** us-west-2
- **Lambda Function:** *ItemsFunction*

Remember, when you simply select the Lambda Function name, API Gateway will use its **\$LATEST** version. Using **\$LATEST** is generally not recommended since you always want to bind your API Resources to stable Lambda aliases (dev, stage, etc.). The next instruction shows how to map each API Gateway stage to the correct Lambda alias.

3. To use the dev alias, add **:dev** to the end of the **Lambda Function**. The complete value is *ItemsFunction:dev*:

Lambda function

Provide the Lambda function name or alias. You can also provide an ARN from another account.

us-west-2

4. Click **Save**.

5. To update the alias of the Lambda function, click **Edit** in the **Integration request** settings section, and change **:dev** to **:prod**

You are changing the value to give API Gateway permission to the prod alias as well as the dev alias.

6. Click **Save**.

You will now do similar actions for the /items/{ID} resource.

7. Select the **/items > GET > /{ID} > GET** method.

8. Click **Integration Request > Edit**, set the following values and click **Save**

- **Integration type:** Lambda Function
- **Use Lambda Proxy integration:** unchecked
- **Lambda Region:** us-west-2
- **Lambda Function:** *ItemsFunction:dev*
- **Request body passthrough:** When there are no templates defined

This time you need to configure the **Mapping Templates** because the resource includes a dynamic {ID} variable. You will map the ID path parameter to a Lambda event parameter, via a custom mapping template.

9. Scroll down to **Mapping templates**.

10. Enter the following values

- **Content type:** *application/json*
- **Generate template:** Items
- **Template body:**

Copy code

```
{  
  "ID": $input.params("ID")  
}
```

▼ **Mapping templates**

Content type

```
application/json
```

Generate template

```
Items
```

Template body

```
1 ▶ {  
2   "ID": $input.params("ID")  
3 }
```

11. Click **Save**.

12. To update the alias of the Lambda function, click **Edit** in the Integration request settings section, and change **:dev** to **:prod**:

13. Click **Save**.

At this point, all the API Gateway Resources can access both Lambda Aliases and we won't need to manually grant any additional permission. You now need to redeploy the API into the prod stage.

14. Click **Deploy API** and select **prod** for **Stage** before clicking **Deploy**.

If everything went fine, the same endpoint will finally return real data from the Lambda function, instead of fake data.

In theory, you'd need to deploy the dev-configured Resources into the dev Stage and the prod-configured resources into the prod Stage. For now, you can just deploy the same configuration into both stages (dev and prod) since you will completely change the backend configuration in the next step, to make it more dynamic and flexible.

15. Go back to **Resources**, click **Deploy API** and select **dev** for **Stage** before clicking **Deploy**.

Always remember that you can test your API integration in the AWS Console before deploying it (return to **Method Execution** and click **TEST**). This way you will also be shown the corresponding API Gateway logs for debugging purposes.

Summary

In this lab step, you configured the API backend integration to use the Lambda function you created. You deployed the updated API to both the dev and prod stages.

Following Best Practices for Versions, Aliases, and Stages

Introduction

You have previously created the following resources:

- Lambda function (`ItemsFunction`)
- Lambda Versions (`$LATEST` and `1`)
- Lambda Aliases (dev and prod)
- API Gateway Stages (dev and prod)
- API Gateway Resources (`ItemList` and `Item`)

Both the API endpoints are attached to the same Lambda function and they currently use its version `1`. This configuration is not very robust and doesn't allow you to update your Lambda function and see the updates in your dev environment.

The recommended best practice is connecting your API Gateway stages to the corresponding Lambda Alias, so that you can easily manage new functionalities, testing, bug fixing, rollbacks, etc. You can achieve this setup by creating a new *stage variable* and then use this variable as a Lambda alias in your API Gateway backend configuration. Stage Variables can be used to configure the request integration of your resources so that you don't have to modify them every time you want to re-deploy your API.

Instructions

1. In the **dev Stage Editor**, click on the **Stage variables** tab > **Edit** > **Add stage variable**:

Stage variables (0/0) Edit

2. Enter the following values before clicking **Save**:

- **Name:** *lambdaAlias*
- **Value:** *dev*

3. Select the **prod** stage and create a *lambdaAlias* stage variable with the value of *prod*.

Now you can update the API Gateway Resources' configuration to use the stage variable.

4. For both the /items **GET** method and the /items/{ID} **GET** method, modify the **Integration Request Lambda Function** to:

[Copy code](#)

ItemsFunction:\${stageVariables.lambdaAlias}

Integration request settings Edit

Integration type Info Lambda	Region us-west-2
Lambda proxy integration Info False	Lambda function ItemsFunction:\${stageVariables.lambdaAlias}
Paths ID	Input passthrough When there are no templates defined (recommended)

Note: Because the Lambda alias is dynamically configured via a stage variable in API Gateway, API Gateway cannot automatically create the necessary permissions. You would normally

have to manually grant API Gateway permission to invoke the dev and prod Lambda function aliases. API Gateway warns you about this:

You can grant the permission with the AWS CLI. For the lab, you don't need to grant any additional permission because you already granted API Gateway access to the dev and prod aliases that are referenced by the stage variables.

5. Re-deploy your dev and prod stages before proceeding (**Deploy API**).

To test the API, you can no longer use the **Test** button in the **Method Execution diagram** because the method now depends on being executed in the context of a specific stage.

6. Click on **Stages > prod** and copy the **Invoke URL**:

The screenshot shows the 'Stages' section of the AWS API Gateway console. On the left, there's a sidebar with two stages: 'dev' and 'pro'. The 'pro' stage is selected and highlighted with a blue background. To the right, the 'Stage details' for the 'pro' stage are displayed. The 'Stage name' is 'pro'. Under 'Cache cluster', it says 'Provisioned'. Under 'Default method-level caching', it says 'Inactive'. The 'Rate' is set to 10000 and the 'Burst' is set to 5000. There are sections for 'Web ACL' and 'Client certificate', both of which are currently empty. At the bottom, the 'Invoke URL' is listed as <https://cd8fqlrx74.execute-api.us-west-2.amazonaws.com/pro>. There are 'Stage actions' and 'Create stage' buttons at the top right.

7. Paste the **Invoke URL** into a new browser tab and append */items* to the end before navigating to the URL:

Pretty print

```
[  
  {  
    "ID": 1,  
    "name": "Pen",  
    "price": 2.5  
  },  
  {  
    "ID": 2,  
    "name": "Pencil",  
    "price": 1.5  
  },  
  {  
    "ID": 3,  
    "name": "Highlighter",  
    "price": 3  
  },  
  {  
    "ID": 4,  
    "name": "Ruler",  
    "price": 5  
  }  
]
```

The list of items is returned, confirming that the stage variable for the prod stage is configured correctly.

Summary

In this lab step, you configured stage variables to reference the Lambda function used for the API Gateway backend integration. Now that API Gateway is correctly configured you are free to update the Lambda function code and play with its alias mapping without coming back to API Gateway.

Creating API Keys and Usage Plans

2/2

2 out of 2 validations checks passed

Introduction

The API Resources are still *open*, meaning that no authorization is required. Usually, you will need to secure your API and eventually have a granular way to restrict their access.

In this lab step, you will see the most basic authorization method via API Keys, managed by AWS. Please note that API Keys are intended to track API consumers and define custom throttling and rate limiting.

In the API Gateway Console, you can create or import API Keys and associate them with your API Gateway stages. AWS also offers the concept of *usage plans*, which allow you to manage throttling and quotas in a more granular way. Technically, *each* API Key must be bound to one or more Usage Plans, which can be bound to one or more API stages. This way, you can control and monitor each API Key's usage and cluster similar keys together based on your own needs.

Instructions

1. In the API Gateway Console, select **Usage Plans** in the left navigation panel and click **Create usage plan**
2. In the **Create Usage Plan** form set the following values before clicking **Create usage plan**:
 - **Name:** *CloudAcademyPlan*
 - **Description:** *Cloud Academy Lab Plan*
 - **Throttling:** Unchecked
 - **Quota:** Unchecked

Usage plan details
Usage plans specify the number or rate of requests allowed per second.

Name
CloudAcademyPlan

Description - optional
Cloud Academy Lab Plan

Throttling
Limit the rate that users can call your API.

Quota
Turn on quotas to limit the number of requests

You will enable throttling and a quota in a later lab step.

3. Click on the newly created usage plan, select the **Associated stages** tab, click **Add Stage**, and set the following values before clicking **Add to usage plan**
 - **API:** *ItemsAPI*
 - **Stage:** *dev*
4. Click **Add Stage** again and associate the plan to the *ItemsAPIprod* stage as well.

In most situations, you would create a different plan for each stage so that you can also have independent API Keys. Within the scope of this Lab, you can simply bind both stages to the same usage plan.

5. Select the **Associated API Keys** tab and click on **Add API Key**.

6. In the **API Key** form, enter the following values before clicking **Add API key**:

- **Type:** Create and add new key
- **Name:** LabKey
- **Description:** Cloud Academy Lab Key
- **API key:** Generate a key automatically

API key details

Type

- Add existing key
 Create and add new key

Name

LabKey

Description - optional

Cloud Academy Lab Key

API key

- Generate a key automatically
 Provide a custom key

Once the API Key is created, it is bound to the two API Gateway stages in the usage plan.

7. In the list, click **LabKey**, and click **Show** next to **API Key** in the upper section of the panel to reveal the API Key (40-char alphanumeric string):

Make a note of the API Key. You will use this API Key as a custom HTTP header later on. But first, you need to update the API Gateway to require API Keys.

8. For both the **/items** and **/items/{ID}** resource **GET** methods, click **Method Request > Edit** in the diagram, check the **API key Required** box, and click **Save**:

Method request settings

Authorization

None

Request validator

None

 API key required

10. Re-deploy the API to both stages (**Deploy API**).

You won't be able to simply GET the resources in your browser anymore. In that case, since the endpoint is not open anymore, you'd receive a {"message": "Forbidden"} response.

11. (Optional) If you have a system with the curl command available (available on mac OSX and most Linux distributions) you can send an authenticated request using the following command:

[Copy code](#)

```
curl -H "x-api-key: YOUR_API_KEY" YOUR_API_HOST/prod/items/
```

If everything worked fine, you should see the JSON output as follows:

Note: If you don't have the curl command available, you can use a web-based curl site to test the command. Search the web for "web based curl" to find one.

You can create up to 500 API Keys and 300 Usage Plans per account. This means you can't associate one unique API Key to each User of your application. API Keys are more suitable for server-to-server scenarios or 3rd-party customers. Alternatively, you can secure API Gateway with your own custom IAM roles or with custom Lambda Authorizers.

The limits for the Amazon API Gateway service are published by AWS in the [Developer Guide](#), [opens in a new tab](#). Many of the limits can be increased by contacting AWS support.

[Summary](#)

In this lab step, you created a usage plan and associated it with your API stages. You then created an associated API key with the usage plan and required API keys to access the API.

Enabling CORS on API Gateway Resources

Introduction

CORS stands for Cross-origin resource sharing. It is a set of standard HTTP headers used to restrict the access of web resources from other domains. In the web app security model, this is called same-origin policy, and it's supposed to avoid cross-site scripting (XSS) attacks. Browsers will not fetch the requested resources unless the corresponding server attaches the required HTTP headers in the OPTIONS call (or GET method).

The API Gateway will not automatically add these headers, but it provides a very user-friendly wizard. In this Lab Step, you will see how to enable CORS on our API Gateway resources. Keep in mind that you need to enable it on each resource.

Instructions

1. Select the **/items** API Resource and click on **Enable CORS**

This starts a wizard that allows you to configure which headers to serve and which headers are allowed. A minimal CORS integration would require at least the Access-Control-Allow-Origin header. If its value is *, every domain will be allowed to fetch this resource. The API Gateway will automatically allow GET and OPTIONS methods, plus all the default headers used by AWS services (*Content-Type*, *X-Amz-Date*, *X-Api-Key*, *X-Amz-Security-Token* and *Authorization*). This will guarantee that you can use API Keys, Basic Authentication and IAM tokens.

2. Select **GET** under **Access-Control-Allow-Methods**, leave the remaining default values, and click **Save**:

The screenshot shows the AWS API Gateway 'Enable CORS' configuration page. At the top, there is a breadcrumb navigation: API Gateway > APIs > Resources - ItemsAPI (cd8fqrlx74) > Enable CORS. Below the breadcrumb, a green success message box displays the text "Successfully enabled CORS" with a "Details" link. The main configuration area is titled "Enable CORS". It contains three sections: "CORS settings" (with an info link), "Gateway responses" (checkboxes for "Default 4XX" and "Default 5XX"), and "Access-Control-Allow-Methods" (checkboxes for "GET" and "OPTIONS", where "GET" is checked). The "GET" checkbox is highlighted with a blue border. At the bottom, there is a section for "Access-Control-Allow-Headers" with the note "API Gateway will configure CORS for the selected gateway responses." and a list of headers: "Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token".

Note that there is not too much magic going on here. The API Gateway will simply configure itself to enable CORS on the specified resource, by creating a new OPTIONS method and all the required Method Response and Integration Response options.

3. Repeat the above instructions except using the **/items/{ID}** resource to enable CORS for it.

After enabling CORS on both resources, your API should look like this:

The screenshot shows the AWS API Gateway Resource Details page. On the left, the resource tree shows a root node '/' with a child node '/items' containing 'GET' and 'OPTIONS' methods. A selected node is '/(ID)' containing 'GET' and 'OPTIONS' methods. On the right, the 'Resource details' section shows the path '/items/{ID}' and resource ID 'qd1dl8'. It includes buttons for 'Delete', 'Update documentation', and 'Enable CORS'. The 'Methods' section lists two methods: 'GET' (Integration type: Lambda, Authorization: None, API key: Required) and 'OPTIONS' (Integration type: Mock, Authorization: None, API key: Not required). There are also 'Delete' and 'Create method' buttons for the methods list.

Note: The OPTIONS method will be invoked by the browser as a preflight request, before the actual GET request. If the OPTIONS call doesn't succeed, browsers won't even issue your GET request.

Summary

In this lab step, you used the API Gateway wizard to enable CORS on both of your API resources. CORS enhances the security of web apps.

Enabling API Gateway Caching and Throttling

Introduction

You can configure throttling and caching independently for each API Gateway stage. For example, you may want to avoid throttling and caching on your dev stage, and enable them on your prod stage.

Please note that although throttling is free, caching is provided by a custom instance whose size and cost depend on the selected cache capacity (from 500MB to 237GB). Optionally you can encrypt your cache data, set up a time-to-live (TTL) in seconds and require authorization for cache invalidation requests.

You will configure caching and throttling for the prod stage in this Lab Step.

Instructions

1. Select the prod stage, click **Edit** in the **Stage details** section, toggle on **Provision API cache**, and enter the following values:

- **Cache capacity: 0.5GB**
- **Encrypt cache data: Checked**
- **Cache time-to-live (TTL): 3600**
- **Require authorization: Toggled on**
- **Unauthorized request handling: Fail the request with a 403 status code**

Cache settings Info

You can enable API caching to cache your endpoint's responses. With caching, you can reduce latency and save costs. For more information about API caching and its pricing, see API Gateway pricing for details.

Provision API cache

Provision API caching capabilities for your stage. Caching is not active.

Default method-level caching

Activate method-level caching for all GET methods in this stage.

Cache capacity

0.5GB

Encrypt cache data

Cache time-to-live (TTL)

seconds

3600

Must be between 0-3600 seconds.

Per-key cache invalidation

Require authorization

Unauthorized request handling

Fail the request with 403 status code

2. Scroll down to the **Throttling settings** section and ensure that **Throttling** is toggled on and accept the default values

3. Click **Continue > Save changes** at the bottom of the tab.

In case you have special customers and you want to limit their access to your API, you can configure the throttling and daily/weekly/monthly quota on the corresponding Usage Plan.

4. Select **Usage Plans > CloudAcademyPlan > Actions > Edit usage plan** and observe the **Throttling** and **Quota** sections

You can set usage plan-level limits here.

5. Toggle on **Quota** and enter a quota of **1000** requests **Per day** before clicking **Save changes**:

Quota

Turn on quotas to limit the number of requests a user can make to your API in a given time period.

Requests

Enter the total number of requests that a user can make in the time period you select in the dropdown list.

1000

Per day



This configuration will ensure that each individual user (as identified by unique API keys) won't consume your stage/account capacity. Of course, you can monitor and update these configurations at any time, even after creating a Usage Plan or an API Key.

Summary

In this lab step, you enabled API caching and throttling on the prod stage. You also enabled a quota at the usage plan-level to limit the number of requests authorized by an API Key per day.

Cleaning up API Resources and Lambda Functions

0/2

0 out of 2 validations checks passed

Introduction

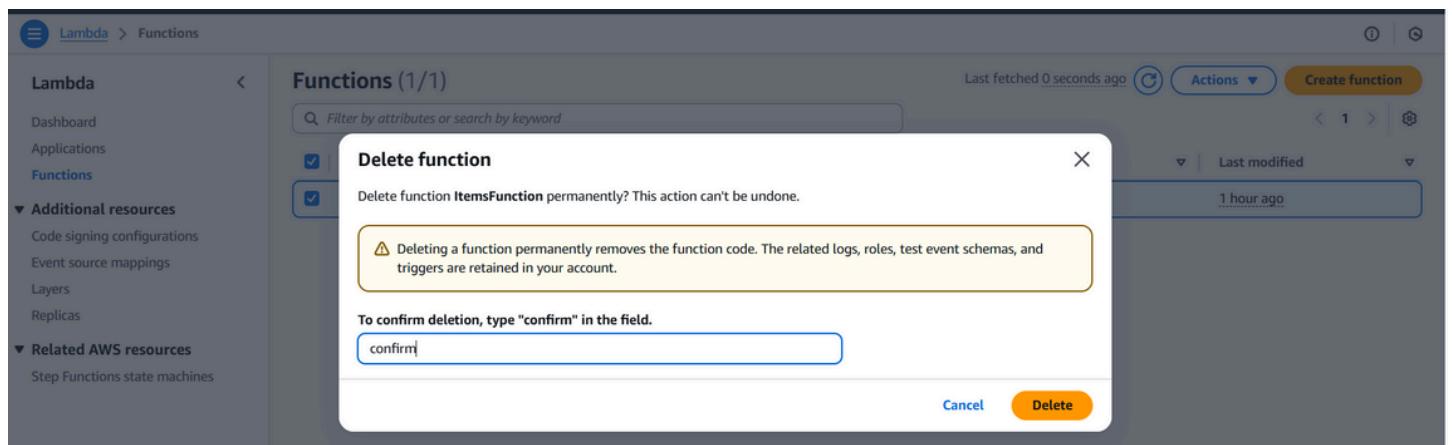
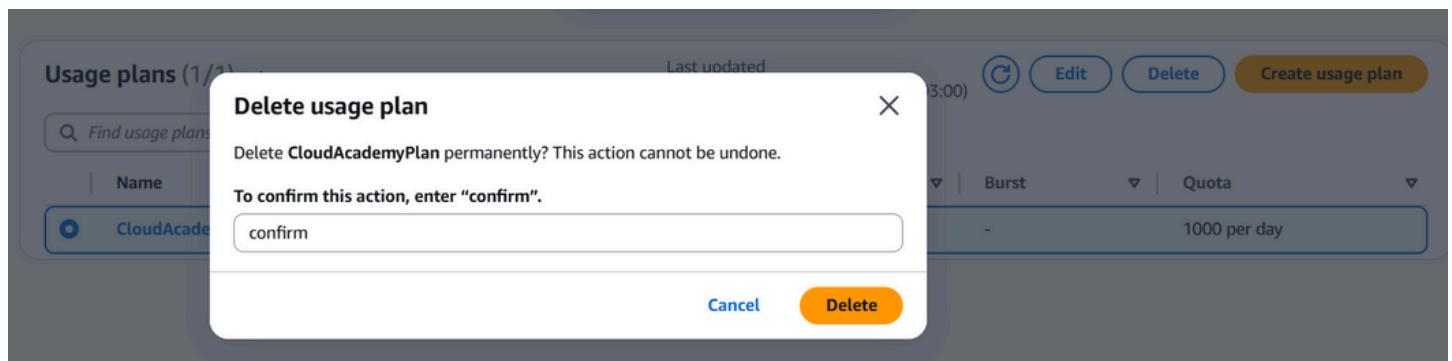
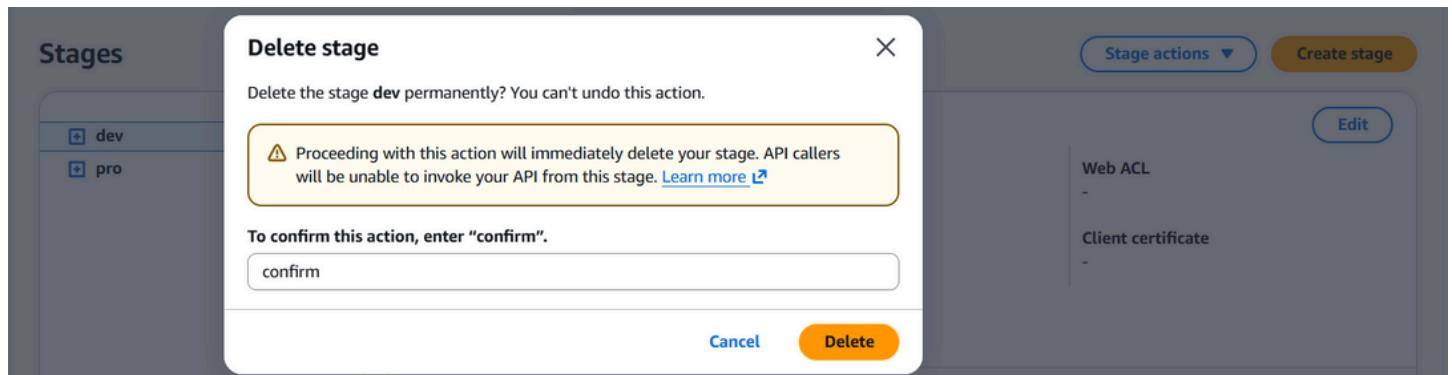
As a final step, you will delete all the resources created in the lab.

Instructions

1. Delete the following resources in the given order to avoid conflicts:

- API Key (**API Keys > Delete API Key**)
- API Stages (**Stages > dev/prod > Delete Stage**)
- Usage Plan (**Actions > Delete Usage Plan**)
- API Resources (**Actions > Delete API**)
- Lambda Function (**Actions > Delete function**)

The screenshot shows the AWS API Gateway 'API keys' page. A success message at the top says 'Successfully updated usage plan 'CloudAcademyPlan''. A modal window titled 'Delete API key' is open, asking if the user wants to permanently delete the 'LabKey' API key. A warning message in the modal states: '⚠️ This API key is currently active. Proceeding with this action will immediately delete your API key. API callers will be unable to use this API key to access your APIs.' At the bottom of the modal are 'Cancel' and 'Delete' buttons. To the right of the modal, a table lists the 'Lab Key' with a creation date of 'November 16, 2025, 21:01 (UTC+03:00)'.



Please note that you won't be able to delete Usage Plans if at least one API Stage is bound to it. That's why you'll need to delete stages first.

Alternatively, you can just delete the API altogether with its stages. You will need to confirm the API name in order to delete every API resource, model, stage, stage variable, etc.

By deleting the Lambda Function, you will also delete all the related versions and aliases.

Interestingly, you could also delete the Lambda Function first and the AWS Console will not complain, although API Gateway will not have a backend anymore and your API will return a 500 error (*Internal Server Error*).

Code source [Info](#)

[Open in Visual Studio Code](#) [Upload from](#)

EXPLORER

- ITEMSFUNCTION
 - lambda_function.py

lambda_function.py

```

1 # static list of items
2 items = [
3     {"ID": 1, "name": "Pen", "price": 2.5},
4     {"ID": 2, "name": "Pencil", "price": 1.5},
5 ]

```

PROBLEMS **OUTPUT** **CODE REFERENCE LOG** **TERMINAL**

Status: Succeeded
Test Event Name: GetItems

Response:

```

[
    {
        "ID": 1,
        "name": "Pen",
        "price": 2.5
    },
    {
        "ID": 2,
        "name": "Pencil",
        "price": 1.5
    }
]

```

DEPLOY [Current](#)

- Deploy (Ctrl+Shift+U)
- Test (Ctrl+Shift+I)

TEST EVENTS [SELECTED: GETITEMS]

- + Create new test event
- Private saved events

Create new test event

Event Name:
Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Execution Results

Activate Windows
Go to Settings to activate Windows

[Code](#) [Test](#) [Monitor](#) [Configuration](#) [Aliases](#) [Versions](#)

Aliases (2) [Info](#)

Last fetched 2 minutes ago

Name	Versions	Description
dev	version: \$LATEST (weight=100%)	Development alias
Prod	version: 11 (weight=100%)	Production alias

[Edit](#) [Delete](#) [Create alias](#)

[API Gateway](#) > [APIs](#) > [ItemsAPI \(cd8fqrx74\)](#) > [Stages](#) > Manage stage variables

Successfully updated stage variables for stage 'dev'.

Manage stage variables

Stage variables

Name	Value	Action
lambdaAlias	prod	Remove

[Add stage variable](#)

[Cancel](#) [Save](#)

[API Gateway](#) > [Usage plans](#) > [CloudAcademyPlan](#) > Associate stage

Successfully created usage plan 'CloudAcademyPlan'.

Associate stage [Info](#)

Stage details

API: ItemsAPI

Stage: dev

You must select an API before selecting a stage.

Method-level throttling - optional

[Cancel](#) [Add to usage plan](#)

Elastic Kubernetes Service IAM S3 EC2 CloudWatch API Gateway Aurora and RDS DynamoDB VPC Athena

API Gateway > Usage plans > CloudAcademyPlan > Add API key

Add API key

API key details

Type
 Add existing key
 Create and add new key

Name

Description - optional

API key
 Generate a key automatically
 Provide a custom key

[Cancel](#) [Add API key](#)

API key details

ID	I97nwfxxpl	Status	✓ Active
Description	Cloud Academy Lab Key	API key Show
Creation date	November 16, 2025, 21:01 (UTC+03:00)		

[Associated usage plans](#) [Tags](#)

Usage plans (1)

Last updated	November 16, 2025, 21:01 (UTC+03:00)	Edit	Remove
Usage plan ID	Name	Associated APIs	Associated stages
<input type="radio"/> 7w6e7y	CloudAcademyPlan	ItemsAPI ItemsAPI	dev pro

API Gateway > APIs > [ItemsAPI \(cd8fqrlrx74\)](#) > Stages > Edit stage

Attribute	Current value	New value
Cache cluster enabled	<input type="radio"/> Inactive	<input checked="" type="radio"/> Active
Encrypt cache data	-	<input checked="" type="radio"/> Active
Cache time-to-live (TTL)	-	3600 seconds
Per-key cache invalidation	-	<input checked="" type="radio"/> Active
Unauthorized request handling	-	Fail the request with 403 status code
Throttling	<input type="radio"/> Inactive	<input checked="" type="radio"/> Active
Cache capacity	-	0.5 GB
Throttle rate	-	10000 requests per second
Throttle burst	-	5000 requests

⚠ When you save your changes, these changes take effect immediately. Changing the settings of your stage might affect how API callers use your API.

Edit usage plan

Usage plan details

Usage plans specify the number or rate of requests that your API accepts from a client. Associate an API key with a usage plan to track the requests your API receives.

Name

CloudAcademyPlan

Description - optional

Cloud Academy Lab Plan

Throttling

Limit the rate that users can call your API.

Quota

Turn on quotas to limit the number of requests a user can make to your API in a given time period.

Requests

Enter the total number of requests that a user can make in the time period you select in the dropdown list.

1000



Per day

