

Lab 7.1: Creating Lambda Functions Using the AWS SDK for Python

Lab overview and objectives

In this lab, you will use the AWS SDK for Python (boto3) to create AWS Lambda functions. Calls to the REST API that you created in the earlier Amazon API Gateway lab will initiate the functions. One of the Lambda functions will perform either an Amazon DynamoDB database table scan or an index scan. Another Lambda function will return a standard acknowledgment message that you will enhance later in a lab where implement Amazon Cognito.

After completing this lab, you should be able to:

- Create a Lambda function that queries a DynamoDB database table.
- Grant sufficient permissions to a Lambda function so that it can read data from DynamoDB.
- Configure REST API methods to invoke Lambda functions using Amazon API Gateway.

Duration

This lab will require approximately **90 minutes** to complete.

AWS service restrictions

In this lab environment, access to AWS services and service actions might be restricted to the ones that are needed to complete the lab instructions. You might encounter errors if you attempt to access other services or perform actions beyond the ones that are described in this lab.

Scenario

The café is eager to launch a dynamic version of their website so that the website can access data stored in a database. Sofia has been making steady progress toward this goal.

In a previous lab, you played the role of Sofia and created a DynamoDB database. The database table contains café menu details, and an index holds menu items that are flagged as specials. Then, in another lab, you created an API to add the ability for the website to receive mock data through REST API calls.



In this lab, you will again play the role of Sofia. You will replace the mock endpoints with functional endpoints so that the web application can connect to the database. You will use Lambda to bridge the connection between the GET APIs and the data stored in DynamoDB. Finally, for the POST API call, Lambda will return an updated acknowledgment message.

Accessing the AWS Management Console

1. At the top of these instructions, choose to launch your lab.

A **Start Lab** panel opens, and it displays the lab status.

Tip: If you need more time to complete the lab, choose the **Start Lab** button again to restart the timer for the environment.

2. Wait until you see the message *Lab status: ready*, then close the **Start Lab** panel by choosing the **X**.

3. At the top of these instructions, choose .

This opens the AWS Management Console in a new browser tab. The system will automatically log you in.

Tip: If a new browser tab does not open, a banner or icon is usually at the top of your browser with a message that your browser is preventing the site from opening pop-up windows. Choose the banner or icon and then choose **Allow pop ups**.

4. Arrange the AWS Management Console tab so that it displays along side these instructions. Ideally, you will be able to see both browser tabs at the same time so that you can follow the lab steps more easily.

Tip: If you want the lab instructions to display across the entire browser window, you can hide the terminal in the browser panel. In the top-right area, clear the **Terminal** ☐ check box.

Task 1: Configuring the development environment

In this first task, you will configure your AWS Cloud9 environment so that you can create Lambda functions.

5. Connect to the AWS Cloud9 integrated development environment (IDE).
 - o From the **Services** menu, search for and select **Cloud9**.

Notice the existing IDE, which is named **Cloud9 Instance**.

- For that IDE, choose **Open**.

The AWS Cloud9 IDE loads in a new browser tab.

6. Download and extract the files that you will need for this lab.

- In the same terminal, run the following command:

```
wget https://aws-tc-largeobjects.s3.us-west-2.amazonaws.com/CUR-TF-200-ACCDEV-2-91558/05-lab-lambda/code.zip -P /home/ec2-user/environment
```

- The code.zip file is downloaded to the AWS Cloud9 instance. The file is listed in the left navigation pane.
- Extract the file:

```
unzip code.zip
```

7. Run the script that re-creates the work that you completed in earlier labs into this AWS account.

Note: This script populates the Amazon Simple Storage Service (Amazon S3) bucket with the café website code and configures the bucket policy as you did in the Amazon S3 lab. The script also creates the DynamoDB table and populates it with data as you did in the DynamoDB lab. Finally, the script re-creates the REST API that you created in the API Gateway lab.

- To set permissions on the script and then run it, run the following commands:

```
chmod +x ./resources/setup.sh && ./resources/setup.sh
```

- When prompted for an IP address, enter the IPv4 address that the internet uses to contact your computer. You can find this IP address from <https://whatismyipaddress.com>.

Note: The IPv4 address that you set will be used in the bucket policy. Only requests that originate from the IPv4 address that you identify will be allowed to load the website pages. Do not set it to 0.0.0.0, because the S3 bucket's block public access settings will block this address.

8. To verify the SDK for Python is installed, run the following command in the AWS Cloud9 terminal:

```
pip show boto3
```

Note: If you see a message about not using the latest version of pip, ignore the message.

9. Take a moment to see what resources the script created.

- Confirm that an S3 bucket is hosting the café website files:
 - In the *Your environments* browser tab, browse to the Amazon S3 console, and choose the name of the bucket that was created.
 - Choose **index.html** and copy the **Object URL**.
 - Load the URL in a new browser tab.

The café website displays. Currently, the website is accessing the hard-coded menu data that is stored in S3 to display the menu information.

Tip: Notice that several menu items are listed in the **Browse Pastries** section of the page.

- Confirm that DynamoDB has the menu data stored in a table:

- Browse to the DynamoDB console.
- Choose **Tables** and choose the **FoodProducts** table.
- Choose **Explore table items** and confirm that the table is populated with menu data.
- Choose **View table details** and then on the **Indexes** tab, confirm that an index named **special_GSI** was created.
- Confirm that the ProductsApi REST API was defined in API Gateway:
 - Browse to the API Gateway console.
 - Choose the name of the **ProductsApi** API.
 - The API has a **GET** method for **/products** and a **GET** method for **/products/on_offer**.
 - Finally, the API has **POST** and **OPTIONS** methods for **/create_report**.
 - You can use the **TEST** buttons for each method to ensure that they are returning the mock data that you used in the previous lab. Each method should return a 200 HTML status code.

10. Copy the invoke URL for the API to your clipboard.

- In the API Gateway console, in the left panel, choose **Stages** and then choose the **prod** stage.
Note: If you see a warning that you do not have ListWebACLs and AssociateWebACL permissions, ignore the warning.
- Copy the **Invoke URL** value that displays at the top of the page.
You will use this value in the next step.

11. Update the website's config.js file.

- In the AWS Cloud9 IDE browser tab, open resources/website/**config.js**.
- On line 2, replace `null` with the invoke URL value that you copied a moment ago. Also, be sure to surround the URL in double quotation marks.
- The file now looks like the following example, but you will have a different value for `<some-value>`:

```
window.COFFEE_CONFIG = {
  API_GW_BASE_URL_STR: "https://<some-value>.execute-api.us-east-1.amazonaws.com/prod",
  COGNITO_LOGIN_BASE_URL_STR: null
};
```

- Verify that `/prod` appears at the end of the URL with no trailing slash.
- Save the change to the file.

12. Update and then run the update_config.py script.

- Open python_3/**update_config.py** in the text editor.
- Replace the `<FMI_1>` placeholder with the name of your S3 bucket.
Tip: Find the bucket name in the S3 console, or run the following command:

```
aws s3 ls
```

- Notice that this script will upload the config.js file that you just edited to the S3 bucket.
- Save the change to the file.
- To run the script, run the following commands:

```
cd ~/environment/python_3
python update_config.py
```

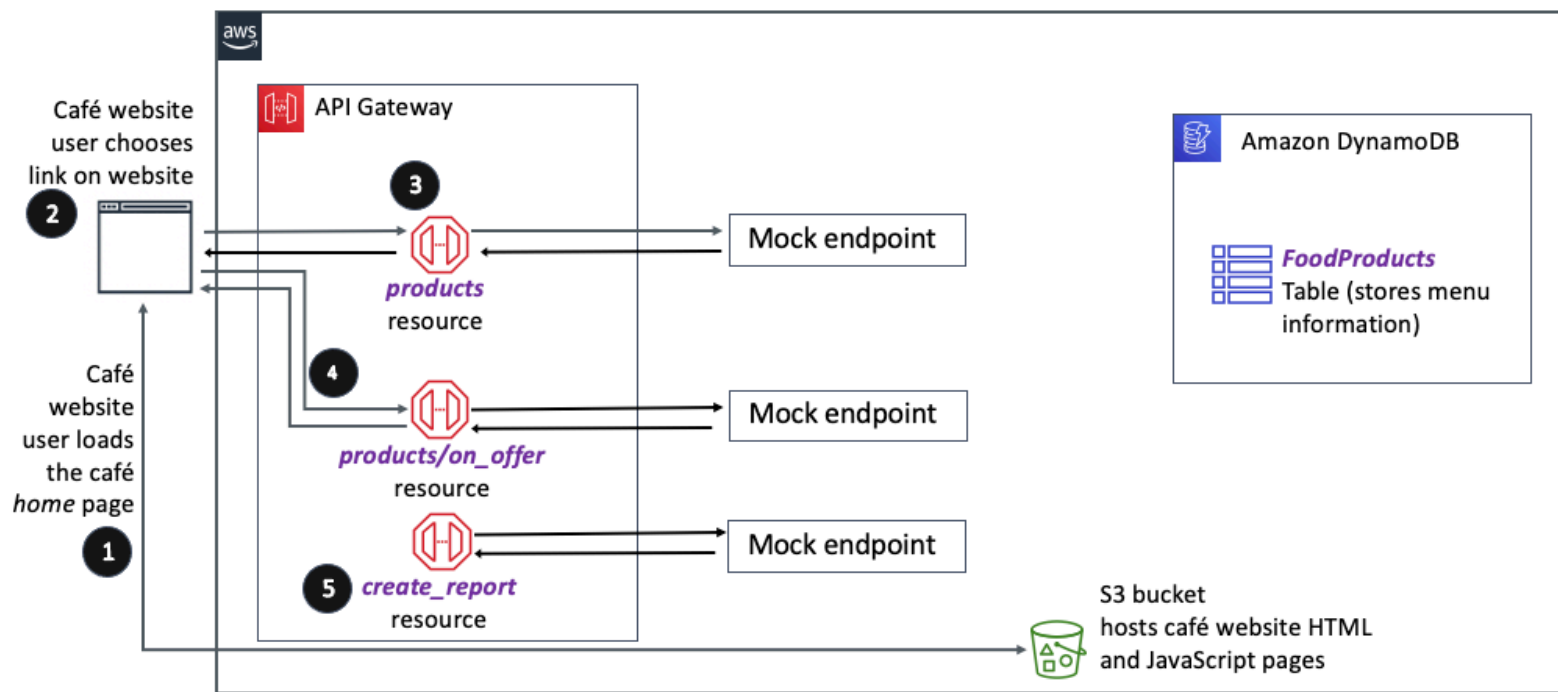
13. Load the latest café webpage with the developer console view exposed.
- If you still have the café website open in a browser tab, return to it. If you do not have the website open, reopen it now by following these steps:
 - In the Amazon S3 console, choose the name of the bucket that contains your website files.
 - Choose **index.html** and then copy the **Object URL** value.
 - Load the object URL in a new browser tab.
 - Refresh the browser tab to load the changes.

The café website displays.

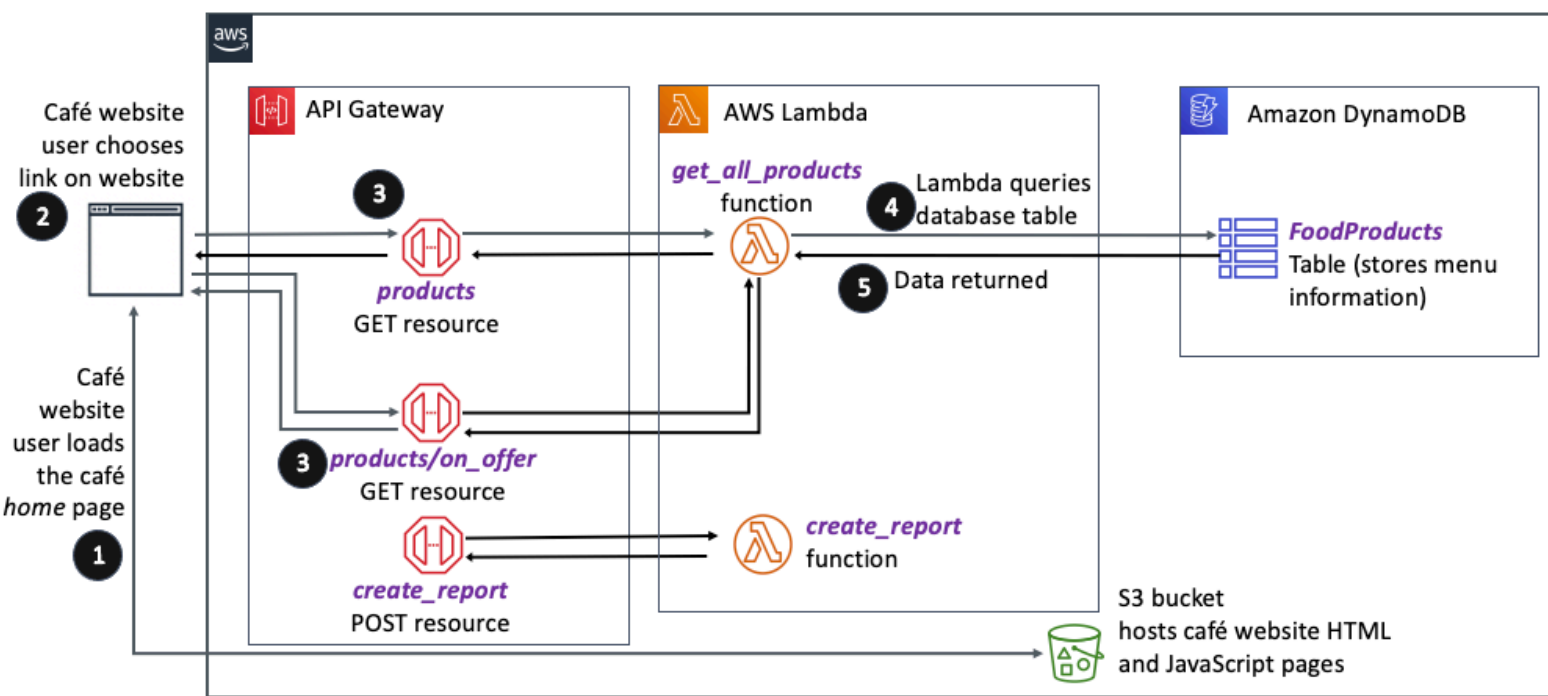
- Observe the **Browse Pastries** section of the webpage. Notice that only one menu item now displays.

This indicates that you are no longer viewing hard-coded data. Instead, the website returns the mock data just as you configured it to do in the previous lab.

Your AWS account is now configured as shown in the following diagram:



By the end of this lab, you will have created Lambda functions that the API will invoke. At that point, your account resources and configurations will look like the following diagram:



Task 2: Creating a Lambda function to retrieve data from DynamoDB

The first Lambda function that you create will respond to any `/products` GET requests from the website. The function will replace the mock endpoint that you created for the products API resource in the previous lab.

14. Observe and edit the Python code that you will use in the Lambda function.

- In the AWS Cloud9 file browser, browse to and open `python_3/get_all_products_code.py`.
- Replace the `<FMI_1>` placeholder and the `<FMI_2>` placeholder with the proper values.

Tip: To find the missing values in the code, return to the DynamoDB console.

- Notice that the code does the following:
 - It creates a boto3 client to interact with the DynamoDB service.
 - It reads the items out of the table and returns the menu data.
 - It also scans the table index and filters for items that are in stock.
- Save the changes to the file.

15. Test the code *locally* in AWS Cloud9.

- To ensure that you are in the correct folder, run the following command:

```
cd ~/environment/python_3
```

- To run the code locally in the AWS Cloud9 terminal, run the following command:

```
python3 get_all_products_code.py
```

- If the code runs successfully, the first part of the table data is returned in the terminal output, formatted as a JSON document as shown here.


```
running scan on table
{'product_item_arr': [{'price_in_cents_int': 295, 'special_int': 1, 'tag_str_arr': ['doughnut', 'on offer'], 'description_str': 'A doughnut with blueberry jelly filling.', 'product_name_str': 'blueberry jelly doughnut', 'product_id_str': 'a455'}, {'price_in_cents_int': 295, 'tag_str_arr': ['doughnut', 'on offer'], 'description_str': 'so good!', 'product_name_str': 'vanilla glazed doughnut', 'product_id_str': 'a453'}, {'price_in_cents_int': 295, 'tag_str_arr': ['doughnut', 'on offer'], 'description_str': 'Boston's favorite doughnut, done right.', 'product_name_str': 'boston cream doughnut', 'product_id_str': 'a458'}],
...truncated for brevity
```

16. Modify a setting in the code and test it again.

- In the `get_all_products_code.py` file, line 12 has the following code:

```
if offer_path_str is not None:
```

Analysis: If the `offer_path_str` variable is not found, the condition fails and runs a scan of the table.

- To verify that this logic is working, temporarily reverse this condition. Remove the word `not` from this line of code, so that it looks like the following:

```
if offer_path_str is None:
```

- Save the change.
- Run the code again:

```
python3 get_all_products_code.py
```

- The output resembles the following example:

```
running scan on index
{'product_item_arr': [{'price_in_cents_int': 295, 'special_int': 1, 'tag_str_arr': ['doughnut', 'on offer'], 'description_str': 'A doughnut with blueberry jelly filling.', 'product_name_str': 'blueberry jelly doughnut', 'product_id_str': 'a455'}, {'price_in_cents_int': 595, 'special_int': 1, 'tag_str_arr': ['pie slice', 'on offer'], 'description_str': 'A delicious slice of Frank's homemade pie.', 'product_name_str': 'apple pie slice', 'product_id_str': 'a444'}, {'price_in_cents_int': 395, 'special_int': 1, 'tag_str_arr': ['bagel', 'on offer'], 'description_str': 'Boiled in salt water, then baked. As it should be.', 'product_name_str': 'plain bagel', 'product_id_str': 'a465'}],
...truncated for brevity
```

- Notice that fewer menu items were returned this time. This was the intended result. The website calls to the REST API will implement the code so that customers can filter the menu for "on offer" menu items.

Now that you know that the code logic works, reverse the code change and then define the Lambda function that will run the code.

17. Reverse the change that you made to the code.

- On line 12, add the word `not` back to the code so that it again reads as the following:

```
if offer_path_str is not None:
```

18. Comment out the last line of the code and save the file.

- To comment out the last line of code, use a `#`. The line should look like the following:

```
#print(lambda_handler({}, None))
```

- Save the change.

19. Locate the IAM role that the Lambda function will use, and copy the role Amazon Resource Number (ARN) value.

- Browse to the IAM console, and choose **Roles**.
- In the search box search for and select the **LambdaAccessToDynamoDB** role that has been created for you.

Notice that this role provides read only access to DynamoDB. The policy provides enough access for Lambda to read the data that is stored in DynamoDB.

- Copy the **Role ARN** value.

You will use this in the next step.

20. Edit the wrapper code that you will use to create the Lambda function.

- Return to the AWS Cloud9 file browser.
- Browse to and open `python_3/get_all_products_wrapper.py`.
- On line 5, replace `<FMI_1>` with the role ARN value that you copied.
- Save the changes to the file.

21. Observe what the `get_all_products_wrapper.py` code will accomplish when it is run:

- Creates a Lambda boto3 client.
- Sets the name of the role that the Lambda function should use.
- References the location of the S3 bucket that has the code that the Lambda function should run (the code you just zipped and placed in Amazon S3).
- Uses all of this information to create a Lambda function. The function definition identifies Python 3.8 as the runtime.

22. Package the code and store it in an S3 bucket.

- A bucket with `-s3bucket-` in the name was created for you when you started the lab.
- Verify that your AWS Cloud9 terminal is in the **python_3** directory.

```
cd ~/environment/python_3
```

- To place a copy of your code in a .zip file, run the following command:

```
zip get_all_products_code.zip get_all_products_code.py
```

- Next, to retrieve the name of your S3 bucket, run the following command:

```
aws s3 ls
```

- Finally, to place the .zip file in the bucket, run the following command. Replace `<bucket-name>` with the actual bucket name that you retrieved:

```
aws s3 cp get_all_products_code.zip s3://<bucket-name>
```


- Verify that the command succeeded.

The response looks like the following: `upload: ./get_all_products_code.zip to s3://<bucket-name>/get_all_products_code.zip`

23. To create the Lambda function, run the following command:

```
python3 get_all_products_wrapper.py
```

The output of the command shows `DONE`, confirming that the code ran without errors.

24. Observe the function that you created and test it.

- Browse to the Lambda console.
- Choose the name of the **get_all_products** function that you just created.
- In the **Code source** panel, open (double-click) the **get_all_products_code.py** file to display the code.
- Choose **Test**.
- For **Event name**, enter `Products`
- Keep all of the other default test event values, and choose **Save**.

The test event is saved.

- Choose **Test** again.

A tab that shows the results of your test displays, with a response that shows the data returned from the DynamoDB table. The following is an example:

```
{
  "product_item_arr": [
    {
      "price_in_cents_int": 295,
      "special_int": 1,
      "tag_str_arr": [
        "doughnut",
        "on offer"
      ],
      "description_str": "A doughnut with blueberry jelly filling.",
      "product_name_str": "blueberry jelly doughnut",
      "product_id_str": "a455"
    },
    {
      "price_in_cents_int": 295,
      "tag_str_arr": [
        "doughnut",
        "on offer"
      ],
      ... Truncated for brevity
    }
  ]
}
```

25. Create a new test event that is called **onOffer**.

- In the **Code source** panel, open the **Test** menu (choose the arrow icon), and choose **Configure test event**.
- Choose **Create new event**.
 - For **Event name**, enter `onOffer`

- In the code editor panel, replace the existing code with the following:

```
{  
  "path": "on_offer"  
}
```

- Choose **Save**.
- Choose **Test**.

This time, the results only display the items that are on offer and not out of stock.

- Scroll to the bottom of the test results to the function logs. You see the log message `running scan on index`.

As you can see, your Lambda function is now successfully retrieving data from the DynamoDB table. You have also observed that this one Lambda function can be used to return *all* menu items *or* only the ones that are on offer.

Congratulations on achieving this milestone!

The next step is to configure the REST API to invoke this Lambda function whenever anyone requests product data on the website. You will also need to find a way to pass the `path` variable to Lambda when customers choose to view only the on offer products.

Task 3: Configuring the REST API to invoke the Lambda function

First, you tested the code locally in AWS Cloud9 to ensure that it worked. Then, you deployed the code as a Lambda function and tested that it worked as deployed. In this task, you will configure the `/products` and `/products/on_offer` REST API functions to invoke the `get_all_products` Lambda function so that the code can be invoked from the café website.

26. Test the existing GET `/products` resource.

- Browse to the API Gateway console.
- Choose the **ProductsApi** API, and choose the **GET** method for **/products**.
Notice on the right side of the page that the method is still accessing a "Mock Endpoint".
- Choose **Test**, and then choose **Test** at the bottom of the page.
- Verify that the Response Body correctly returns the mock data.

27. Replace the mock endpoint with the Lambda function.

- At the top of the page. Ensure that the **GET** method is still selected under **/products**.
- Choose **Integration Request** and **Edit**:
 - Integration type: **Lambda Function**
 - Lambda Region: **us-east-1**
 - Lambda Function: `get_all_products`
 - Choose **Save**
- Choose **Save**.

Notice on the right side of the page that the method is no longer calling a "Mock Endpoint". Instead, it is calling your Lambda function.

28. Test the `/products` GET API call one more time by selecting **Test**.

The call returns output similar to the following:

```

{
  "product_item_arr": [
    {
      "price_in_cents": 295,
      "special": 1,
      "description": "A doughnut with blueberry jelly filling.",
      "product_name": "blueberry jelly doughnut",
      "product_id_str": "a455",
      "tags": [
        "doughnut",
        "on offer"
      ]
    },
    {
      "price_in_cents": 295,
      "description": "so good!",
      "product_name": "vanilla glazed doughnut",
      "product_id_str": "a453",
      "tags": [
        "doughnut",
        "on offer"
      ]
    }
  ],
  ...truncated for brevity(26 items)
}

```

29. Analyze the results.

- When you switched the integration endpoint from the mock endpoint to the Lambda function, the response headers that permitted Cross-Origin Resource Sharing (CORS) were removed.
- If you scroll down past the Response Data, the **Response Headers** section now shows only the following:

```

{"Content-Type":["application/json"],"X-Amzn-Trace-Id":["Root=1-63066f30-6fcb49923559a3bc52eed2ce;Sampled=0"]}

```

The response header does not contain the CORS information that you need because API Gateway resides in a different subdomain (us-east-1.amazonaws.com) than the S3 bucket (s3.amazonaws.com).

You could manually add the CORS configuration back to this resource using the AWS SDK. However, API Gateway has a feature that makes it simple to permit CORS.

30. Re-enable CORS on the /products API resource.

- Choose **/products** so that it is highlighted.
- Select the **GET** method.
- Select **Default 4XX** and **Default 5XX** under **Gateway responses**.
- Select **GET** under **Access-Control-Allow-Methods**.
- Choose **Save**.

31. Test the /products GET API call one more time.

- Choose the **GET** method for **/products**.
- Choose **Test**, and then choose **Test** at the bottom of the page.
- Scroll down to the **Response Headers** section again.
- This time, `Access-Control-Allow-Origin` is included. The following is an example of the output (your data will have a different value for Root):

```
{"Access-Control-Allow-Origin":["*"],"Content-Type":["application/json"],"X-Amzn-Trace-Id":["Root=1-63066fa3-6e4218d81cb85b1dc0b78d05;Sampled=0"]}
```

32. Using the same approach, update the `/on_offer` GET API method.

- Choose the **ProductsApi** API, and choose the **GET** method for `/on_offer`.
- Choose **Integration Request** and configure:
 - Integration type: **Lambda Function**
 - Lambda Region: **us-east-1**
 - Lambda Function: `get_all_products`
- Choose **Save**.
- Choose `/on_offer` so that it is highlighted.
- Choose **Enable CORS**.
- Select **Default 4XX** and **Default 5XX** under **Gateway responses**.
- Select **GET** under **Access-Control-Allow-Methods**.
- Choose **Save**.

33. Test the `/on_offer` GET API call.

- Choose the **GET** method for `/products/on_offer`.
- Choose **Test**, and then choose **Test** at the bottom of the page.
- Scroll down to the **Response Headers** section. Notice that CORS is enabled in the headers.
- Scroll back up to the **Response Body** section. Do you notice an issue?

The Lambda function is returning all of the menu item, not just the specials.

This is because the conditional check for `on_offer_str` is not working.

Recall the relevant code:

```
offer_path_str = event.get('path')
if offer_path_str is not None:
```

Analysis: You need to set the API logic for `/products/on_offer` to pass the path to the event object that Lambda uses. The next step explains how to do that.

34. Configure the `/on_offer` integration request details.

- Choose the **GET** method for `/products/on_offer`.
- Choose **Integration Request**.
- Choose **Edit**. Expand **Mapping Templates**.
- Choose **Add mapping template**.
- In the Content-Type box, enter the following text:

```
application/json
```

- Under **Generate template** choose **Method request passthrough**.
- Replace the text with the following:

```
{  
  "path": "$context.resourcePath"  
}
```

This will evaluate to `/products/on_offer`. For now, the code simply checks for the existence of the variable.

- Choose **Save** at the bottom of the page.

35. Test the GET method for `/on_offer` again.

- Choose **Test**, and then choose **Test** at the bottom of the page.
- Verify that only the on offer items (six items) are returned.

Excellent! Now that this filter is working, you can deploy the API.

36. Deploy the API.

- In the **Resources** panel, choose the API root `/`.
- Choose **Deploy API**.
- For **Deployment stage**, choose **prod**, and then choose **Deploy**.

Congratulations! You have successfully updated the REST API so that it invokes a single Lambda function that can now filter for on offer items. With this logic, you do not need to create and maintain two separate Lambda functions to handle this functionality.

Task 4: Creating a Lambda function for report requests in the future

In this task, you will complete steps that are similar to what you just did. However, this time you will create the Lambda function for the `/create_report` POST action.

37. Observe and test the Python code that you will use in the Lambda function.

- Back in the AWS Cloud9 IDE, browse to and open `python_3/create_report_code.py`.

Notice that this code does not do much yet. It simply returns a message. In a later lab, you will implement more useful logic to actually create a report; however, this code will suffice for now.

- Run the following command in the terminal:

```
python3 create_report_code.py
```

- The output returned in the terminal looks like the following:

```
{'msg_str': 'Report processing, check your phone shortly'}
```

Notice the capitalized "R" in the word `Report`. The mock data contained a lowercase "r" instead. This difference is how you can know that the website is accessing the Lambda function and not the mock data.

38. Comment out the last line of the code in the `create_report_code.py` file.

- To comment out the last line of code, use a `#`. The line should look like the following:

```
#print(lambda_handler(None, None))
```

- Save the change.

39. Edit the wrapper code that you will use to create the Lambda function.

- Browse to and open `python_3/create_report_wrapper.py`.
- On line 5, replace the `<FMI_1>` placeholder with the LambdaAccessToDynamoDB **Role ARN** value.
Tip: You may need to return to the IAM console to copy the Role ARN value.
- Save the changes to the file.

40. Package the code and store it in the S3 bucket.

- To place a copy of your code in a .zip file, run the following command:

```
zip create_report_code.zip create_report_code.py
```

- To place the .zip file in the bucket, run the following command. Replace `<bucket-name>` with the actual bucket name:

```
aws s3 cp create_report_code.zip s3://<bucket-name>
```

- Verify that the command succeeded.

The following is the output: `upload: ./create_report_code.zip to s3://<bucket-name>/create_report_code.zip`

41. Finally, to create the Lambda function, run the following command:

```
python3 create_report_wrapper.py
```

The output of the command is `DONE`, confirming that the code ran without errors.

42. Observe the `create_report` function that you created and test it.

- Browse to the Lambda console.
- Choose the name of the **create_report** function that you just created.
- In the **Code source** panel, open (double-click) the **create_report_code.py** file to display the code.
- Choose **Test**.
- For **Event name**, enter `ReportTest`
- Keep all of the other default test event values, and choose **Save**.

The test event is saved.

- Choose **Test** again.

A tab that shows the results of your test displays, with a response that shows the message hardcoded into the function code. The following is an example:

```
{
  "msg_str": "Report processing, check your phone shortly"
}
```

Note the capitalized "R" instead of the lowercase "r" that is used in the mock data. Therefore, you know that the website is accessing the Lambda function and not the mock data.

Task 5: Configuring the REST API to invoke the Lambda function to handle reports

Recall that in a previous task you configured both GET methods in your API to invoke the first Lambda function that you created. In this task, you will complete similar steps to configure the POST method in the API to invoke the new create_report Lambda function.

43. Test the existing POST method for /create_report.

- Browse to the API Gateway console.
- Choose the **ProductsApi** API, and choose the **POST** method for **create_report**.

Notice on the right side of the page that the method is still accessing a "Mock Endpoint".

- Choose **Test**, and then choose **Test** at the bottom of the page.

Verify that the **Response Body** correctly returns the mock data (note the lowercase "r" in the results), as in the following example:

```
{
  "message_str": "report requested, check your phone shortly."
}
```

44. Replace the mock endpoint with the Lambda function.

- Ensure that the **POST** method is still selected.
- Choose **Integration Request** and **Edit**:
 - Integration type: **Lambda Function**
 - Lambda Region: **us-east-1**
 - Lambda Function: `create_report`
 - Choose **Save**.

Notice on the right side of the page that the POST method is no longer calling a "Mock Endpoint". Instead, it is calling the Lambda function.

- Test the method again.

The returned data looks like the following example. Note that you are now seeing a capitalized "R" instead of the lowercase "r" from the mock data:

```
{"msg_str": "Report processing, check your phone shortly"}
```

Note: You could configure CORS on this method as you did for the other API methods. However, you do not need to because the website will not invoke this URL until a later lab when you enable authentication.

45. Deploy the API.

- In the **Resources** panel, choose the API root `/`.
- Choose **Deploy API**.
- For **Deployment stage**, choose **prod**, and then choose **Deploy**.

Congratulations! You have successfully updated the REST API so that it invokes the create_report Lambda function.

Task 6: Testing the integration using the café website

In this final task, you will test both API calls (/products and /products/on_offer) through the website.

46. Load the café website.

- Return to the browser tab where you have the café website open, and refresh the page.
 - **Note:** To find the page again, browse to the Amazon S3 console. Choose the bucket name, choose **index.html**, and copy the **Object URL** value. Load the URL in a new browser tab.

The café website displays. The website is now accessing the menu data that is stored in DynamoDB.

47. Test the menu items filter on the website.

- Scroll down to the **Browse Pastries** section of the page. By default, only the "on offer" menu items display.
- Choose **view all** and verify that more menu items are returned.

If everything displays correctly, that means that your CORS configuration is working properly.

48. Edit a menu item price in the DynamoDB table, and verify that the change is reflected on the website.

- Select your favorite "on offer" menu item and note the current price.
- In a different browser tab, go to the DynamoDB console and load the **FoodProducts** table items.
- To open the menu item's record, choose the **product_name** hyperlink for that item.
- Change the **price_in_cents** value to a different three-digit or four-digit number.
- Save the change.
- Reload the café website, and verify that the price change is reflected on the website.

Update from the café

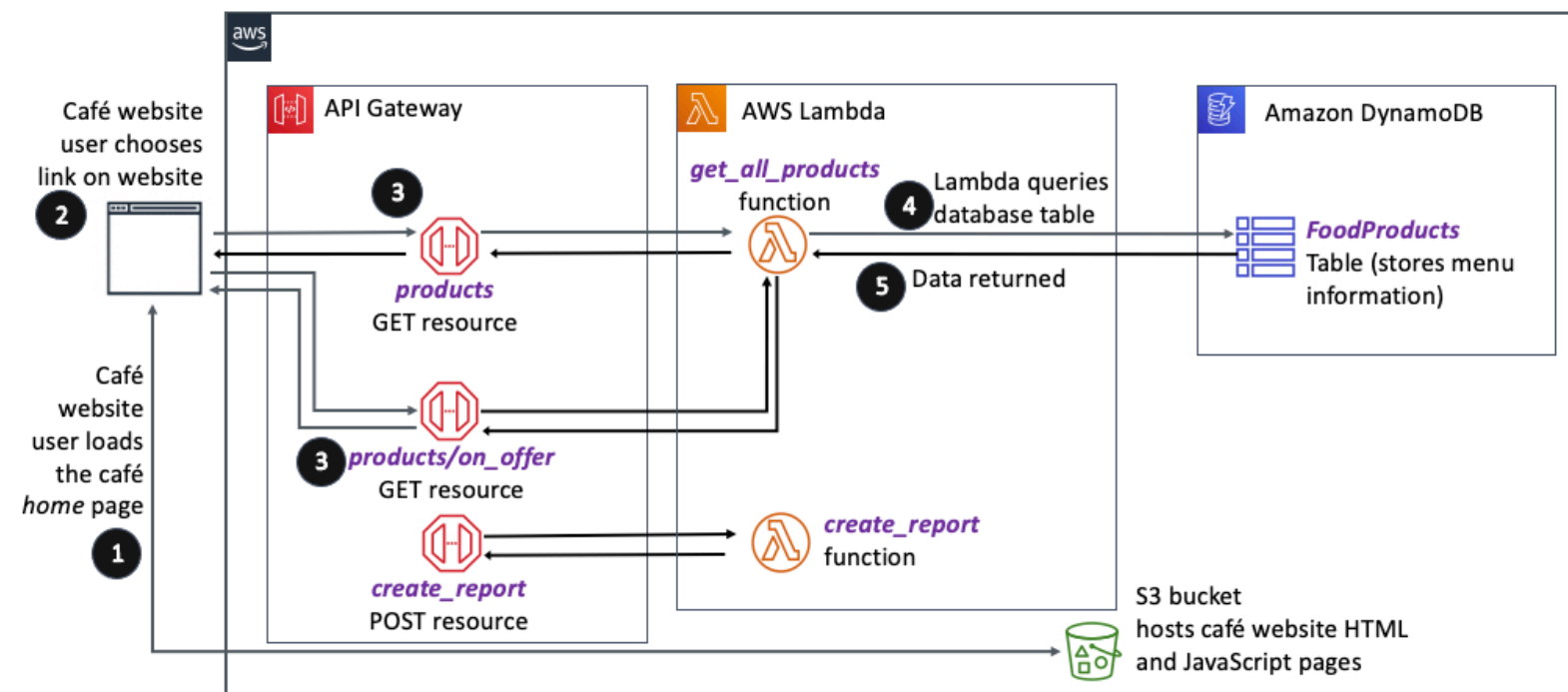


Sofia is satisfied that she has made progress!

Her plan to build a serverless, dynamic website with a database backend is really coming to fruition. Sofia's initial plan had three major milestones:

- The *first milestone* was to create a *database backend* to store café data. She accomplished that in the DynamoDB lab.
- The *second milestone* was to create a *REST API* so that the webpage hosted on Amazon S3 could interact with mock data. She completed the most difficult part of that task during the API Gateway lab.
- The *third milestone* was to create Lambda functions to query the DynamoDB table or index, or return a static message, and then to update the REST API resources to initiate the respective Lambda functions.

You (acting as Sofia) have now built all of these resources in the labs up to this point in the course. The following diagram shows the architecture that you have built.



Sofia knows that she has more work to do. For example, she needs to add authentication so that logged-in users can leverage the `create_report` API and generate reports. That will be implemented behind the login button that appears at the top of the website. Sofia also just heard from the café owners that they are in the final stages of acquiring another company (a coffee bean supplier). The acquisition will probably create additional requirements.

For now though, Sofia decides to celebrate her most recent accomplishment by relaxing with her friends.



Submitting your work

49. At the top of these instructions, choose **Submit** to record your progress and when prompted, choose **Yes**.


Tip: If you previously hid the terminal in the browser panel, expose it again by selecting the **Terminal** ☐ checkbox. This action will ensure that the lab instructions remain visible after you choose **Submit**.

50. If the results don't display after a couple of minutes, return to the top of these instructions and choose **Grades**

Tip: You can submit your work multiple times. After you change your work, choose **Submit** again. Your last submission is what will be recorded for this lab.

51. To find detailed feedback on your work, choose **Details** followed by ► **View Submission Report**.

Lab complete

 Congratulations! You have completed the lab.

52. Choose **End Lab** at the top of this page, and then select **Yes** to confirm that you want to end the lab.

A panel indicates that *DELETE has been initiated... You may close this message box now.*

53. Select the **X** in the top-right corner to close the panel.

