



Build CRUD Microservice with HTTP API and Lambda

Overview:

In this hands-on lab, we'll create a serverless product API using AWS services such as **API Gateway**, **Lambda**, and **DynamoDB**. The API will allow us to perform **CRUD** (Create, Read, Update, Delete) operations on product data.

Architecture:

1. **HTTP API (API Gateway)** will handle incoming requests (GET, POST, DELETE) from the client.
2. **Lambda function** will process the request, interact with **DynamoDB** to perform the CRUD operations, and return a response to the client via API Gateway.
3. **DynamoDB** will store the product data.

API Operations:

- **GET /products**: Retrieve all products.
- **GET /products/{id}**: Retrieve a product by its ID.
- **POST /products**: Create a new product.
- **DELETE /products/{id}**: Delete a product by its ID.

Steps:

1. **Create Lambda Function:**
 - Open the AWS Lambda Management Console.
 - Create a new Lambda function named ProductFunction.
 - Select the runtime (e.g., Node.js, Python) and set permissions.
 - Initially, deploy the function with a simple "Hello from Lambda" response to test the setup.
2. **Log Incoming Requests:**
 - Modify the Lambda function to log incoming event data (JSON) to understand the request format sent by API Gateway. This will help us process incoming requests correctly.
3. **Create HTTP API in API Gateway:**
 - Set up an HTTP API in API Gateway to handle requests.
 - Configure the routes (GET, POST, DELETE) to trigger the Lambda function.
4. **DynamoDB Integration:**

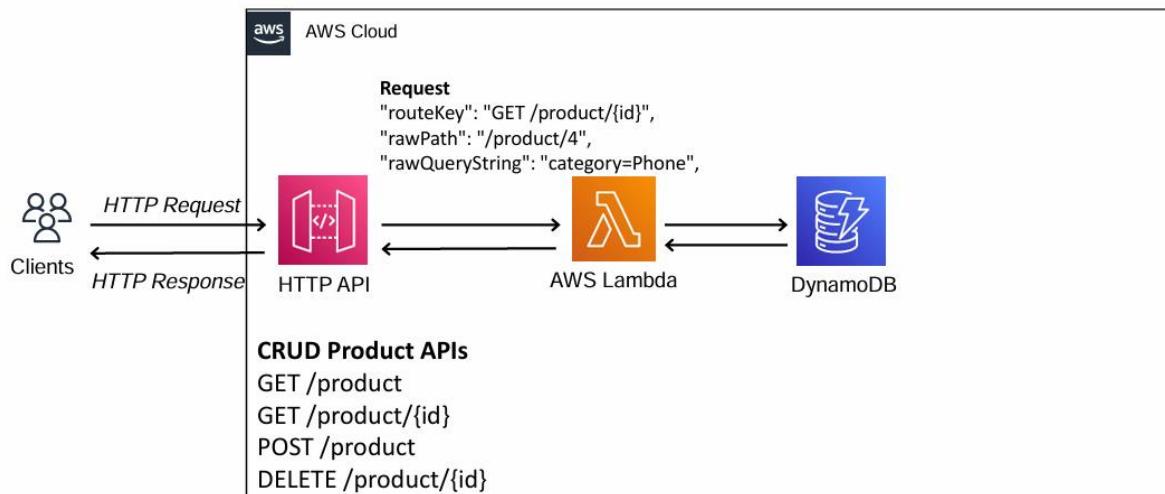
- The Lambda function will interact with DynamoDB to store and retrieve product data.
- We'll set up the necessary DynamoDB tables to store product information (e.g., ProductID, ProductName, etc.).

5. Deploy and Test:

- Deploy the Lambda function with the necessary code to handle CRUD operations.
- Test the API endpoints by sending requests (GET, POST, DELETE) and ensuring the Lambda function processes them and interacts with DynamoDB correctly.

Conclusion:

By the end of this lab, you'll have a fully working serverless API that can handle product data CRUD operations, all running on AWS using **API Gateway**, **Lambda**, and **DynamoDB**.



CRUD stands for the four basic operations that can be performed on data in a database or application. The acronym stands for:

1. **Create**: Add new data or records.
2. **Read**: Retrieve or view existing data.
3. **Update**: Modify or update existing data.
4. **Delete**: Remove existing data.

These operations are fundamental for interacting with databases and are commonly used in building web applications, APIs, and systems that manage data.

Example:

In the context of a **Product API** (as described in your lab):

- **Create**: Add a new product to the database.

- **Read:** Get the details of a product, either all products or a specific product by ID.
- **Update:** Modify the details of an existing product (e.g., change the price or description).
- **Delete:** Remove a product from the database.

CRUD operations are typically mapped to HTTP methods in web APIs:

- **POST** → Create
- **GET** → Read
- **PUT/PATCH** → Update
- **DELETE** → Delete

These operations are the building blocks for managing data in most applications.

To begin with the Lab:

1. First, we will create our Lambda function. In your AWS console, navigate to lambda and click on create.
2. Here you need to choose author from scratch and then give a name to your lambda function. Choose your preferred runtime we'll be choosing Node.js.
3. Then keep everything to default and create your lambda function.

Basic information

Function name
Enter a name that describes the purpose of your function.

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

Runtime [Info](#)
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
 ▼ C

Architecture [Info](#)
Choose the instruction set architecture you want for your function code.
 x86_64
 arm64

Permissions [Info](#)
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▶ [Change default execution role](#)

4. Once your function is created, go inside and change the code with the code below. Click on deploy.

```

export const handler = async (event) => {
  // Log the incoming event object (API Gateway event)
  console.log("event:", JSON.stringify(event, undefined, 2));

  // TODO: Implement your logic here (CRUD operations, interacting with
  DynamoDB, etc.)

  // Return a simple response with a statusCode and body
  const response = {
    statusCode: 200, // HTTP status code (200 means OK)
    body: JSON.stringify('Hello from Lambda!'), // Body of the response
  };

  // Return the response object
  return response;
};

```

```

JS index.mjs ×
JS index.mjs > ...
1  export const handler = async (event) => {
2    // Log the incoming event object (API Gateway event)
3    console.log("event:", JSON.stringify(event, undefined, 2));
4
5    // TODO: Implement your logic here (CRUD operations, interacting with
6    DynamoDB, etc.)
7
8    // Return a simple response with a statusCode and body
9    const response = {
10      statusCode: 200, // HTTP status code (200 means OK)
11      body: JSON.stringify('Hello from Lambda!'), // Body of the response
12    };
13
14    // Return the response object
15    return response;
16  };

```

- Now we are going to move forward and create our HTTP API. For that we need to search and go to the API gateway. From there we need to choose HTTP API and click on Build.

HTTP API

Build low-latency and cost-effective REST APIs with built-in features such as OIDC and OAuth2, and native CORS support.

Works with the following:
Lambda, HTTP backends

Import

Build

6. From step 1 you need to click on Add Integration.

Create an API

Create and configure integrations

Specify the backend services that your API will communicate with. These are called integrations. For a Lambda integration, API Gateway invokes the Lambda function and responds with the response from the function. For HTTP integration, API Gateway sends the request to the URL that you specify and returns the response from the URL.

Integrations (0) [Info](#)

[Add integration](#)

API name
An HTTP API must have a name. This name is cosmetic and does not have to be unique; you will use the API's ID (generated later) to programmatically refer to this API.

[Cancel](#) [Review and Create](#) [Next](#)

7. Then for the integration we need to choose Lambda and choose our lambda function then give a name to our API and click on Next.

Create and configure integrations

Specify the backend services that your API will communicate with. These are called integrations. For a Lambda integration, API Gateway invokes the Lambda function and responds with the response from the function. For HTTP integration, API Gateway sends the request to the URL that you specify and returns the response from the URL.

Integrations (1) [Info](#)

Lambda [Remove](#)

AWS Region Lambda function Version [Learn more.](#)

[Add integration](#)

API name
An HTTP API must have a name. This name is cosmetic and does not have to be unique; you will use the API's ID (generated later) to programmatically refer to this API.

[Cancel](#) [Review and Create](#) [Next](#)

8. In step 2, we are going to configure routes as you can see below in the snapshot, we have to use some methods and you need to use the same methods to configure your routes.

Configure routes - *optional*

Configure routes Info

API Gateway uses routes to expose integrations to consumers of your API. Routes for HTTP APIs consist of two parts: an HTTP method and a resource path (e.g., GET /pets). You can define specific HTTP methods for your integration (GET, POST, PUT, PATCH, HEAD, OPTIONS, and DELETE) or use the ANY method to match all methods that you haven't defined on a given resource.

Method	Resource path	Integration target	
GET	/product	→ ProductFunction	<input type="button" value="Remove"/>
GET	/product/{id}	→ ProductFunction	<input type="button" value="Remove"/>
POST	/product	→ ProductFunction	<input type="button" value="Remove"/>
DELETE	/product/{id}	→ ProductFunction	<input type="button" value="Remove"/>
<input type="button" value="Add route"/>			

Cancel

9. Now we are in the default stage and keep the stage name as it is. Move to review page and create your API.

Define stages - *optional*

Configure stages Info

Stages are independently configurable environments that your API can be deployed to. You must deploy to a stage for API configuration changes to take effect, unless that stage is configured to autodeploy. By default, all HTTP APIs created through the console have a default stage named \$default. All changes that you make to your API are autodeployed to that stage. You can add stages that represent environments such as development or production.

Stage name	Auto-deploy	
\$default	<input checked="" type="checkbox"/>	<input type="button" value="Remove"/>
<input type="button" value="Add stage"/>		

Cancel

10. Once your API is created open it and you will see the details for it.

product-api

API details

API ID ksxst5ugq8	Protocol HTTP	Created 2024-11-14
Description No Description	Default endpoint Enabled https://ksxst5ugq8.execute-api.us-east-1.amazonaws.com	ARN arn:aws:apigateway:us-east-1::apis/ksxst5ugq8

Stages for product-api (1)

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
\$default	https://ksxst5ugq8.execute-api.us-east-1.amazonaws.com	Sir5zv	enabled	2024-11-14

11. Also, you can go to the routes and here you will see that the configuration routes that you defined while creating it.

API Gateway

APIs

Custom domain names

VPC links

API: product-api...(iuxvpuvggk)

Develop

Routes (highlighted with a red box)

Authorization

Integrations

CORS

Reimport

Export

API Gateway > APIs > Routes - product-api (iuxvpuvggk)

Routes

Routes for product-api [Create](#)

Search

- ▼ /product
 - POST
 - GET
- ▼ /{id}
 - DELETE
 - GET

12. Now it is time to test our API. For that go to the API product and from the stages copy the Invoke URL.

APIs

Custom domain names

VPC links

API: product-api... (iuxvpuvggk)

▼ Develop

Stages for product-api (1)

Stage name	Invoke URL
\$default	https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com

13. You need to append this invoke URL with /product and you will see that you are getting hello from lambda. So, this means that our API response is working properly.

← ⏪ 🔍 <https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product>

"Hello from Lambda!"

14. Now we are going to test our lambda function for all the test events. So, to do that open the Post Man tool on your laptop.
15. Then you need to paste the invoke URL using the GET method in your postman workspace and click on Send. You will see that you get a hello from lambda which means that it succeeded.

GET https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product

HTTP <https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product> Save Share ↗

GET https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product Send

Params Authorization Headers (5) Body Scripts Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Body Cookies Headers (5) Test Results 200 OK 577 ms 196 B ⌂ ⌂ ⌂

Pretty Raw Preview Visualize Text ⌂

```
1 "Hello from Lambda!"
```

16. Now you can go to CloudWatch to open the log stream for your lambda function and here you will see the logs generated when you invoked it using the GET method in postman.

```

▼ 2024-11-14T16:28:55.853+05:30 2024-11-14T10:58:55.853Z 1e643321-3ce8-4a13-a2fb-71f7fccdcfa2 INFO event:

2024-11-14T10:58:55.853Z 1e643321-3ce8-4a13-a2fb-71f7fccdcfa2 INFO event:
{
  "version": "2.0",
  "routeKey": "GET /product",
  "rawPath": "/product",
  "rawQueryString": "",
  "headers": {
    "accept": "*/*",
    "accept-encoding": "gzip, deflate, br",
    "cache-control": "no-cache",
    "content-length": "0",
    "host": "iuxvpuvvgk.execute-api.us-east-1.amazonaws.com",
    "postman-token": "823e107e-022f-4429-acbd-0f0e702be6d9",
    "user-agent": "PostmanRuntime/7.42.0",
    "x-amzn-trace-id": "Root=1-6735d7ef-1ffc5c6d79a73c1231feaa3a",
    "x-forwarded-for": "54.86.50.139",
    "x-forwarded-port": "443",
    "x-forwarded-proto": "https"
  },
  "requestContext": {
    "accountId": "878893308172",
    "apiId": "iuxvpuvvgk",
    "domainName": "iuxvpuvvgk.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "iuxvpuvvgk",
    "http": {
      "method": "GET",
      "path": "/product",
      "protocol": "HTTP/1.1",
      "sourceIp": "54.86.50.139",
      "userAgent": "PostmanRuntime/7.42.0"
    }
  }
}

```

17. Then we created a new request for the GET method again in Postman but this time we appended the URL with an ID which is 4 here and then added a key and a value then click on Send.
18. As expected, you can see that we got the response.

The screenshot shows the Postman interface with the following details:

- Request URL:** https://iuxvpuvvgk.execute-api.us-east-1.amazonaws.com/product/4?category=phone
- Method:** GET
- Params:** category (Value: phone)
- Body:** Text (Content: "Hello from Lambda!")
- Response Status:** 200 OK
- Response Headers:** Content-Type: application/json; charset=UTF-8
- Response Body:** "Hello from Lambda!"

19. Similarly, you can go to CloudWatch and view the latest log events.



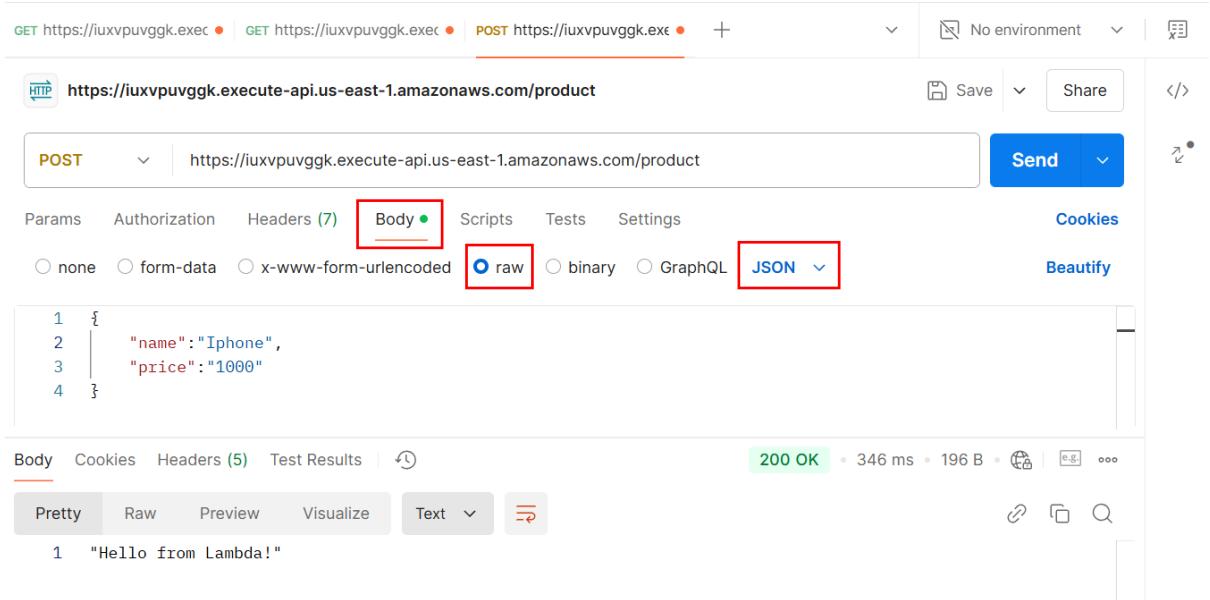
A screenshot of the AWS CloudWatch Logs interface showing a log entry. The log entry timestamp is 2024-11-14T11:05:05.803+05:30, and the log stream ID is b3d24167-1539-42fc-ab2b-86e69573aecc. The log message is INFO event: { "version": "2.0", "routeKey": "GET /product/{id}"}. The log entry details the request headers and query parameters, including 'category': 'phone'.

```

2024-11-14T11:05:05.803+05:30    2024-11-14T11:05:05.803Z b3d24167-1539-42fc-ab2b-86e69573aecc INFO event: { "version": "2.0", "routeKey": "GET /product/{id}"}
2024-11-14T11:05:05.803Z    b3d24167-1539-42fc-ab2b-86e69573aecc    INFO    event:
{
  "version": "2.0",
  "routeKey": "GET /product/{id}",
  "rawPath": "/product/4",
  "rawQueryString": "category=phone",
  "headers": {
    "accept": "*/*",
    "accept-encoding": "gzip, deflate, br",
    "cache-control": "no-cache",
    "content-length": "0",
    "host": "iuxvpuvggk.execute-api.us-east-1.amazonaws.com",
    "postman-token": "7d03c90b-a62d-4367-ab42-ebee4874b263",
    "user-agent": "PostmanRuntime/7.42.0",
    "x-amzn-trace-id": "Root=1-6735d961-7976b17745f65bd0546af68b",
    "x-forwarded-for": "54.86.50.139",
    "x-forwarded-port": "443",
    "x-forwarded-proto": "https"
  },
  "queryStringParameters": {
    "category": "phone"
  }
}

```

20. In the end we created a Post method and as you can see, we got the response from lambda. As you can see in the snapshot for Post you have to choose body then choose raw and change it to JSON.



A screenshot of the Postman application. The request URL is https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product. The method is POST. The request body is set to 'raw' and 'JSON'. The body content is a JSON object with 'name' and 'price' fields. The response status is 200 OK, with a response time of 346 ms and a response size of 196 B. The response body contains the string "Hello from Lambda!".

```

POST https://iuxvpuvggk.execute-api.us-east-1.amazonaws.com/product
Body (raw) JSON
1 {
2   "name": "Iphone",
3   "price": "1000"
4 }

200 OK
1 "Hello from Lambda!"

```

21. If you go and check your CloudWatch logs you will see an event Post method too.

```

        },
        "requestContext": {
            "accountId": "878893308172",
            "apiId": "iuxvpuvggk",
            "domainName": "iuxvpuvggk.execute-api.us-east-1.amazonaws.com",
            "domainPrefix": "iuxvpuvggk",
            "http": {
                "method": "POST",
                "path": "/product",
                "protocol": "HTTP/1.1",
                "sourceIp": "54.86.50.139",
                "userAgent": "PostmanRuntime/7.42.0"
            },
            "requestId": "B08Y7guBIAMEMGA=",
            "routeKey": "POST /product",
            "stage": "$default",
            "time": "14/Nov/2024:11:10:23 +0000",
            "timeEpoch": 1731582623179
        },
        "body": "{\r\n    \"name\": \"Iphone\", \r\n    \"price\": \"1000\"\r\n}",
        "isBase64Encoded": false
    }
}

```

22. Now we are going to update the function code with the code given below and run the methods again in Postman.
23. So, go to your lambda function and update code then deploy it.

```

// Use ES Module (ESM) syntax to define the handler
export const handler = async (event) => {
    console.log('Received event:', JSON.stringify(event, null, 2)); // Log incoming
    event for debugging
    let body;

    try {
        // Switch based on the route key to determine the operation
        switch (event.routeKey) {
            case "GET /product":
                body = `Processing Get All Products`; // Handle GET request to fetch all
                products
                break;
            case "GET /product/{id}":
                if(event.pathParameters != null) {
                    body = `Processing Get Product Id with
                    ${event.pathParameters.id}`; // Handle GET request for a specific product by
                    ID
                }
                break;
            case "POST /product":

```

```

        let payload = JSON.parse(event.body); // Parse incoming JSON body for
POST request
        body = `Processing Post Product Id with "${payload}"`; // Handle POST
request to create a new product
        break;
    case "DELETE /product/{id}":
        if(event.pathParameters != null) {
            body      =  `Processing      Delete      Product      Id      with
"${event.pathParameters.id}"`; // Handle DELETE request for a specific product
by ID
            }
            break;
        default:
            throw new Error(`Unsupported route: "${event.routeKey}"`); // If route
is not supported, throw an error
        }

console.log(body); // Log the body for debugging

return {
    statusCode: 200, // Return HTTP 200 OK for successful operations
    body: JSON.stringify({
        message: `Successfully finished operation: "${event.routeKey}"`,
        body: body
    })
};

} catch (e) {
    console.error(e); // Log any error encountered
    return {
        statusCode: 400, // Return HTTP 400 Bad Request for errors
        body: JSON.stringify({
            message: "Failed to perform operation.",
            errorMsg: e.message // Return the error message for debugging
        })
    };
}

```

24. Now you need to go to Postman and run the methods again. This time you will get the proper response.

GET https://iuxvpuvvggk.execute-api.us-east-1.amazonaws.com/product

Send

Params Authorization Headers (5) Body Scripts Tests Settings Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description	...	

Body Cookies Headers (5) Test Results 200 OK • 1275 ms • 277 B •

Pretty Raw Preview Visualize Text

```
1 {"message": "Successfully finished operation: \"GET /product\"", "body": "Processing Get All Products"}
```

GET https://iuxvpuvvggk.execute-api.us-east-1.amazonaws.com/product/4?category=phone

Send

Params Authorization Headers (5) Body Scripts Tests Settings Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	category	phone		...	
	Key	Value	Description	...	

Body Cookies Headers (5) Test Results 200 OK • 295 ms • 291 B •

Pretty Raw Preview Visualize Text

```
1 {"message": "Successfully finished operation: \"GET /product/{id}\", \"body\": \"Processing Get Product Id with \"4\"\""}  
```

The image consists of three vertically stacked screenshots of the Postman application interface, demonstrating the execution of a CRUD API.

Screenshot 1: POST Request

- Request URL:** https://iuxpuvvggk.execute-api.us-east-1.amazonaws.com/product
- Method:** POST
- Body (JSON):**

```

1 {
2   "name": "Iphone",
3   "price": "1000"
4 }
```

- Response:** 200 OK | 301 ms | 302 B | [Raw](#)
- Response Body (Pretty JSON):**

```

1 {"message": "Successfully finished operation: \"POST /product\"","body": "Processing Post Product Id with \"[object Object]\""}
```

Screenshot 2: DELETE Request

- Request URL:** https://iuxpuvvggk.execute-api.us-east-1.amazonaws.com/product/4
- Method:** DELETE
- Response:** 200 OK | 312 ms | 297 B | [Raw](#)
- Response Body (Pretty JSON):**

```

1 {"message": "Successfully finished operation: \"DELETE /product/{id}\"","body": "Processing Delete Product Id with \"4\""}
```

25. From the above snapshots we can say that our CRUD API is running successfully.
26. Once you are done with the lab delete all the resources. Your Lambda function and your HTTP API.