

Introduction

Serverless computing simplifies application development by removing the need to manage servers. Using AWS Lambda, API Gateway, and DynamoDB, you can create a scalable, cost-effective REST API for tasks like adding and retrieving user data. This setup lets developers focus on code, not infrastructure.

Overview:

AWS Lambda: Executes code in response to events like HTTP requests without managing servers. In this project, it handles adding and retrieving user data from DynamoDB.

API Gateway: Acts as the HTTP interface for Lambda, triggering functions based on requests (POST/GET). No server management is required.

DynamoDB: A fast, fully managed NoSQL database that stores user data and allows easy querying based on user IDs.

Key Features

AWS Lambda: Event-driven, scales automatically, and is cost-efficient with pay-per-use pricing.

API Gateway: Integrates with Lambda, offers security features, and simplifies REST API setup.

DynamoDB: High availability, fast queries, and fully managed—perfect for user data retrieval.

Application:

AWS Lambda and API Gateway create a serverless REST API that scales based on traffic, with DynamoDB providing efficient data storage and retrieval. Together, they eliminate server management and reduce costs, ideal for handling various workloads.

3rd year Project Relation

For the Offsync project, which collects data offline and syncs it once online, this architecture is ideal. Lambda and API Gateway ensure seamless syncing, while DynamoDB handles data storage and retrieval efficiently. The pay-per-use model further optimizes costs for offline-heavy scenarios, and built-in security features protect data.

Step by Step Execution:

Dynamo DB:

Navigate to DynamoDB dashboard>Tables>Click on create table.

I have named the table as Users

Partition key of Users = UserID (string)

[DynamoDB](#) > [Tables](#) > **Create table**

Create table

Table details [Info](#)
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name
This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

1 to 255 characters and case sensitive.

Sort key - optional
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

1 to 255 characters and case sensitive.

Click on create table and you will see this:

[DynamoDB](#) > **Tables**

Tables (1) [Info](#)

<input type="checkbox"/>	Name ▲	Status ▼	Partition key ▼	Sort key ▼	Indexes ▼	Deletion protection ▼	Favorite ▼	Read capacity mode ▼	W
<input type="checkbox"/>	Users	Active	UserID (S)	-	0	Off	☆	Provisioned (5)	Pr

Lambda Functions

Navigate to Lambda dashboard.

AddUserFunction

Click on create function and the following details.

[Lambda](#) > [Functions](#) > **Create function**

Create function [Info](#)

Choose one of the following options to create your function.

☒ **Author from scratch**
Start with a simple Hello World example.

☐ **Use a blueprint**
Build a Lambda application from sample code and configuration presets for common use cases.

☐ **Container image**
Select a container image to deploy for your 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.

[↻](#)

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

☒ **x86_64**

☐ arm64

Keep everything default and click on create function.

Navigate to configuration, then select Permissions to note the execution role name

Code | **Test** | **Monitor** | **Configuration** | **Aliases** | **Versions**

General configuration

Triggers

Permissions

Destinations

Function URL

Execution role

Role name
[addUser-role-503dc9qf](#) [↻](#)

Resource summary

Now navigate to the code section and use this (https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/javascript_dynamodb_code_examples.html) as a reference to write your code logic to put user's data in the table.

Here's my code:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
```

```
// Lambda handler function
```

```
export const handler = async (event) => {
  // Extract parameters from the incoming event (from API Gateway)
  const { userID, name, email } = JSON.parse(event.body);
```

```
  const command = new PutCommand({
    TableName: "Users",
    Item: {
      UserID: userID,
      Name: name,
      Email: email,
    },
  });
```

```
  // Insert data into DynamoDB
  const response = await docClient.send(command);
```

```
  // Return a response for API Gateway
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: "User added successfully",
      data: response,
    }),
  };
};
```

Updating the function addUser.

Code source Info

File Edit Find View Go Tools Window Test Deploy Changes not deployed

Go to Anything (Ctrl-P)

Environment

addUser - /

index.mjs

```
1 import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
2 import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
3
4 const client = new DynamoDBClient({});
5 const docClient = DynamoDBDocumentClient.from(client);
6
7 // Lambda handler function
8 export const handler = async (event) => {
9   // Extract parameters from the incoming event (from API Gateway)
10  const { userID, name, email } = JSON.parse(event.body);
11
12  const command = new PutCommand({
13    TableName: "Users",
14    Item: {
15      UserID: userID,
16      Name: name,
17      Email: email,
18    },
19  });
20
21  // Insert data into DynamoDB
22  const response = await docClient.send(command);
23
24  // Return a response for API Gateway
25  return {
26    statusCode: 200,
27    body: JSON.stringify({
28      message: "User added successfully",
29      data: response,
30    }),
31  };
32 };
33
```

The tests are done in the IAM role config test section.

GetUserFunction

Now create another function from the Lambda function dashboard.
I named it getUser.

Lambda > Functions > Create function

Create function Info

Choose one of the following options to create your function.

☒ Author from scratch

Start with a simple Hello World example.

☐ Use a blueprint

Build a Lambda application from sample code and configuration presets for common use cases.

☐ Container image

Select a container image to deploy for your function.

Basic information

Function name Info

Enter a name that describes the purpose of your function.

getUser

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.

Node.js 20.x

Architecture Info

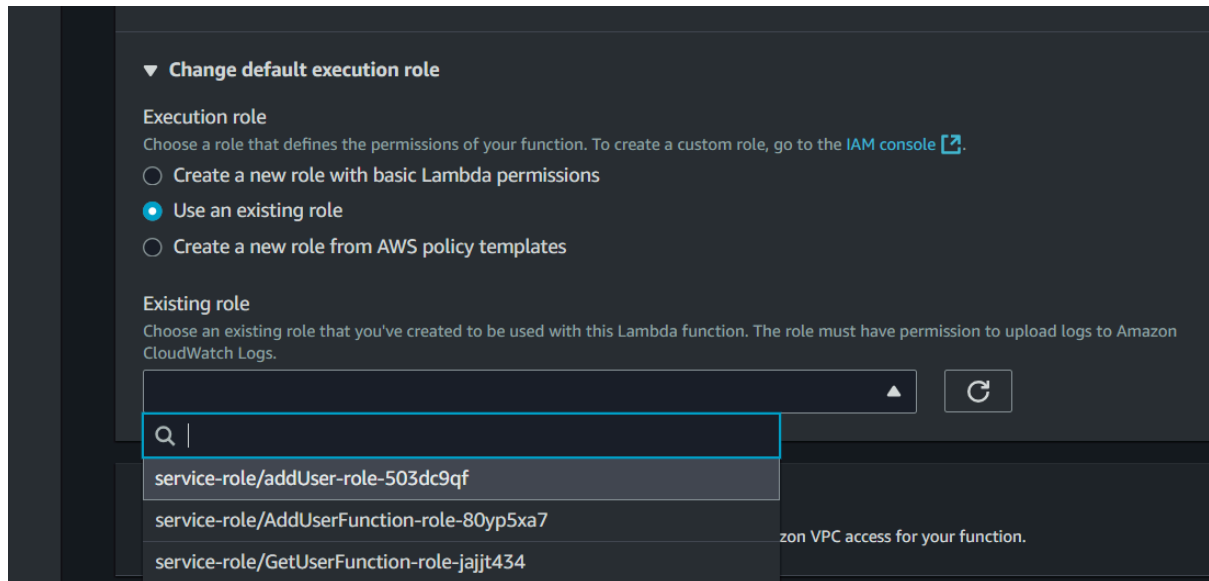
Choose the instruction set architecture you want for your function code.

☒ x86_64

☐ arm64

Permissions

Keep everything default except this time select an existing role which we noted earlier.



Now in the code section use this documentation

(https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/javascript_dynamodb_code_examples.html) as reference and add the logic of getting the user.

Here's my **code**:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
```

```
// Lambda handler function
export const handler = async (event) => {
  // Extract UserID from the query string parameters of the API Gateway event
  const { userID } = event.queryStringParameters;
```

```
  const command = new GetCommand({
    TableName: "Users",
    Key: {
      UserID: userID, // Use the UserID from the event
    },
  });
```

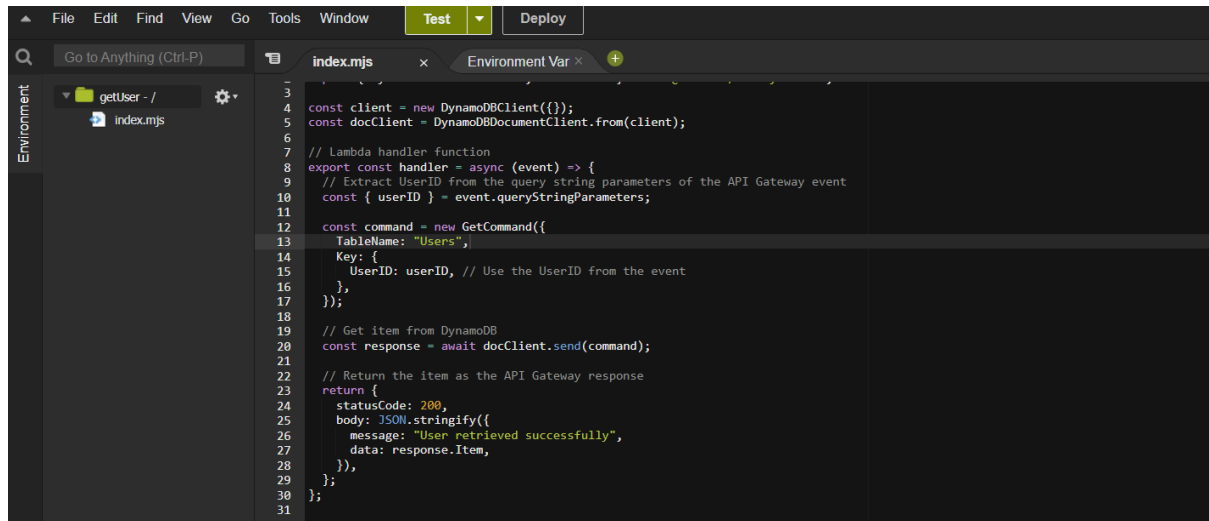
```
  // Get item from DynamoDB
  const response = await docClient.send(command);
```

```
  // Return the item as the API Gateway response
  return {
    statusCode: 200,
    body: JSON.stringify({
```

```

    message: "User retrieved successfully",
    data: response.Item,
  }},
};
};

```



```

1  const client = new DynamoDBClient({});
2  const docClient = DynamoDBDocumentClient.from(client);
3
4  // Lambda handler function
5  export const handler = async (event) => {
6    // Extract UserID from the query string parameters of the API Gateway event
7    const { userID } = event.queryStringParameters;
8
9    const command = new GetCommand({
10     tableName: "Users",
11     Key: {
12       UserID: userID, // Use the UserID from the event
13     },
14   });
15
16   // Get item from DynamoDB
17   const response = await docClient.send(command);
18
19   // Return the item as the API Gateway response
20   return {
21     statusCode: 200,
22     body: JSON.stringify({
23       message: "User retrieved successfully",
24       data: response.Item,
25     }),
26   };
27 };
28
29
30
31

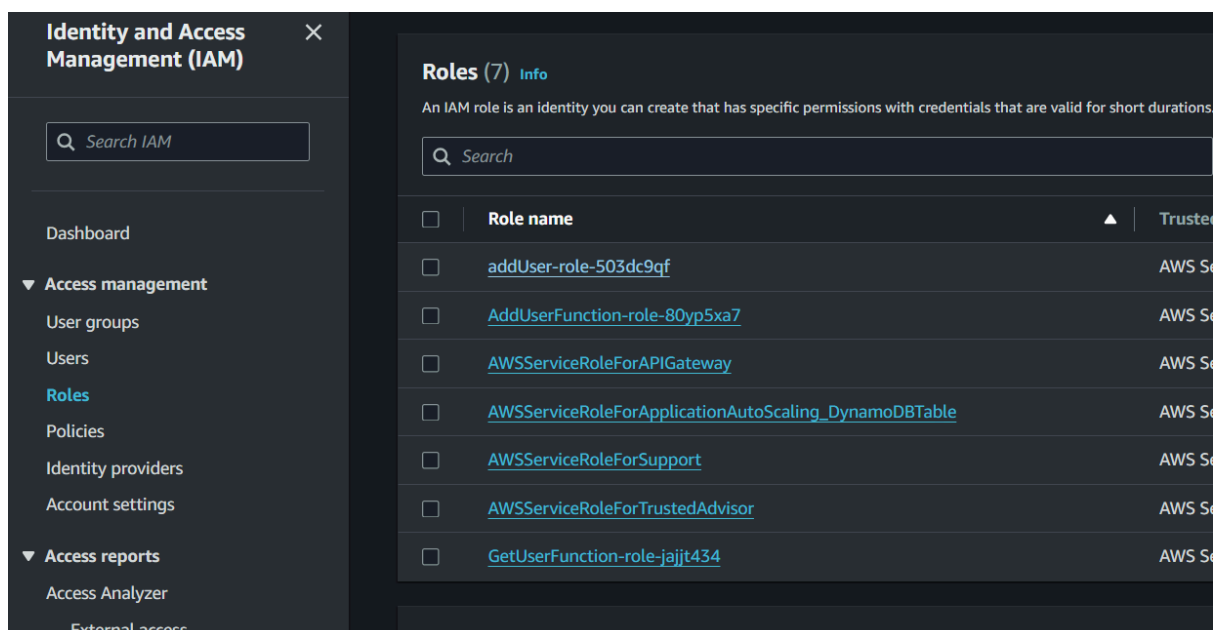
```

The testing is done in the Test Lambda section in the IAM role config.

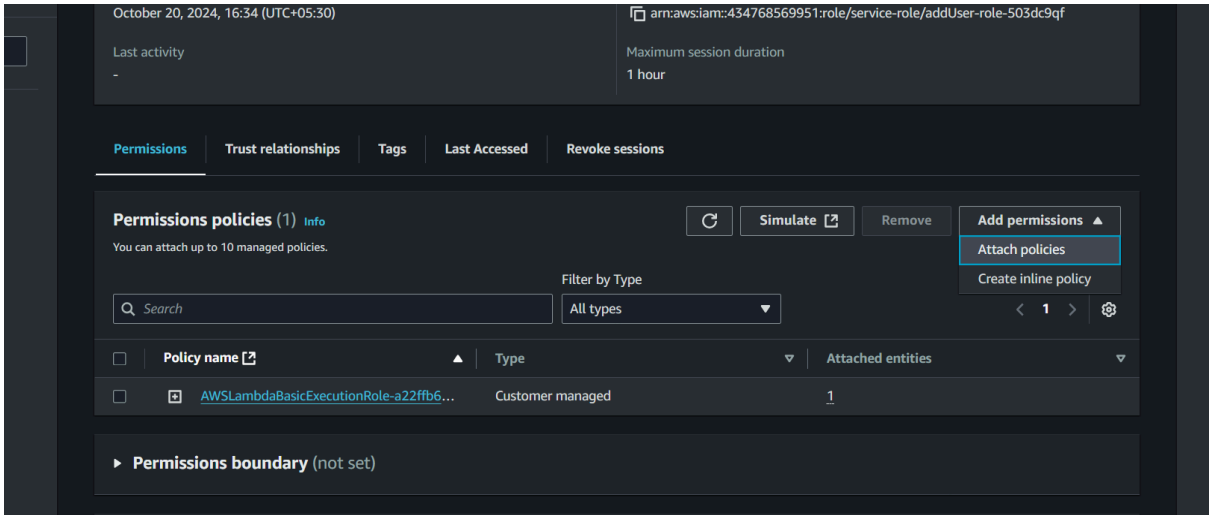
IAM role config

Navigate to IAM roles. Find the role which got created automatically when creating the adduser function and used in both of the lambda functions.

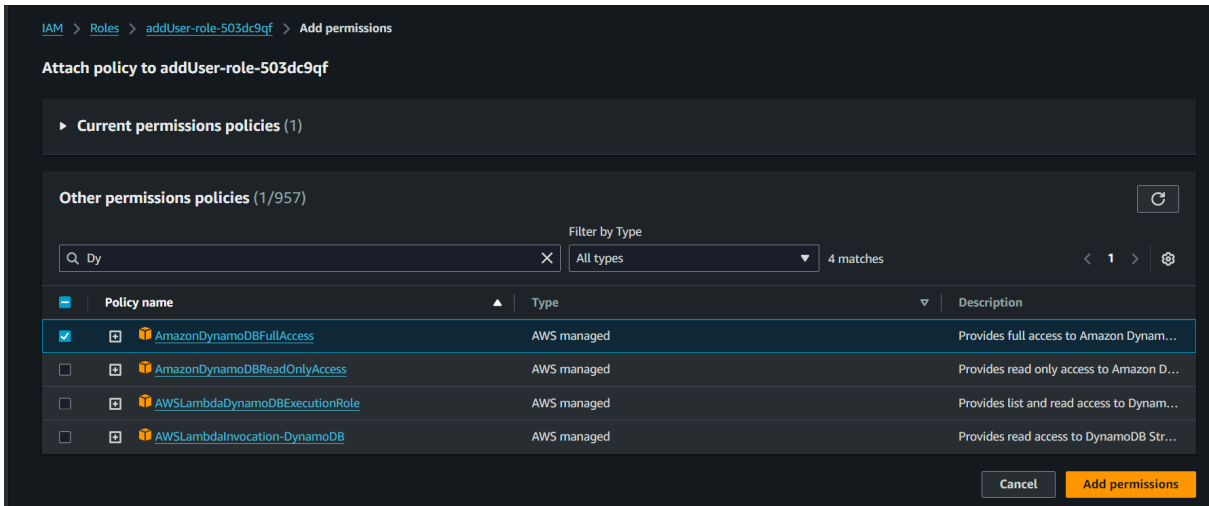
Click on that role:



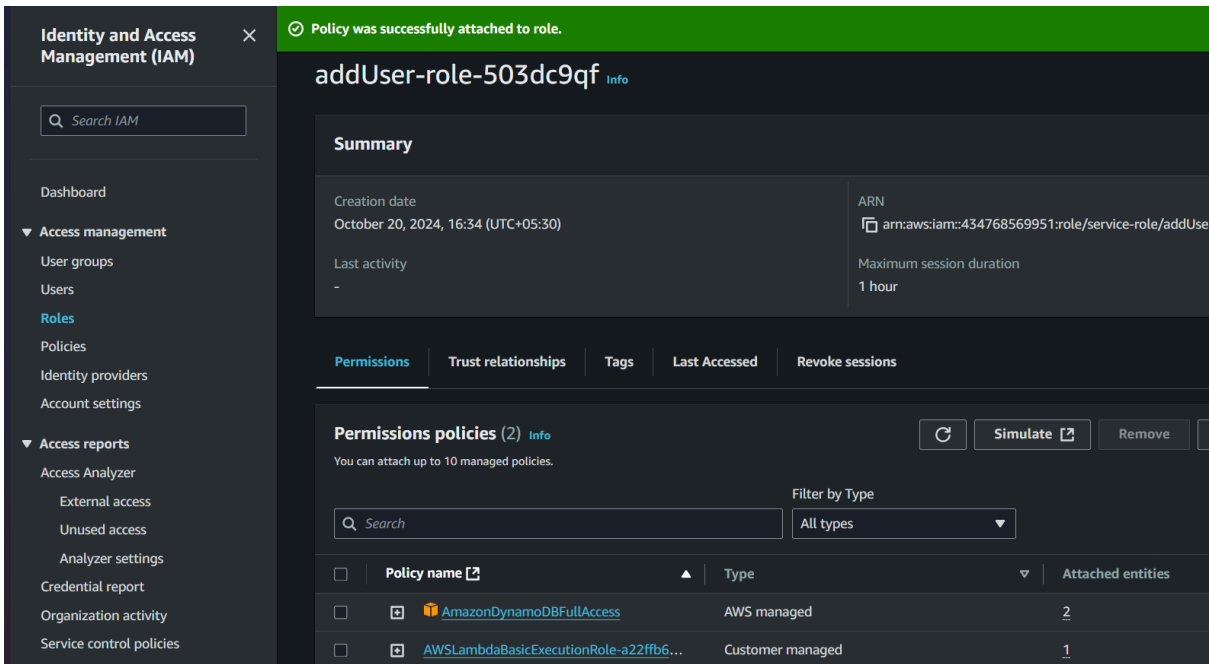
Click on add permissions>attach policies



Select AmazonDynamoDBFullAccess and click on add permissions.



Now you will see two policies attached to the role



Test Lambda Section:

addUser:

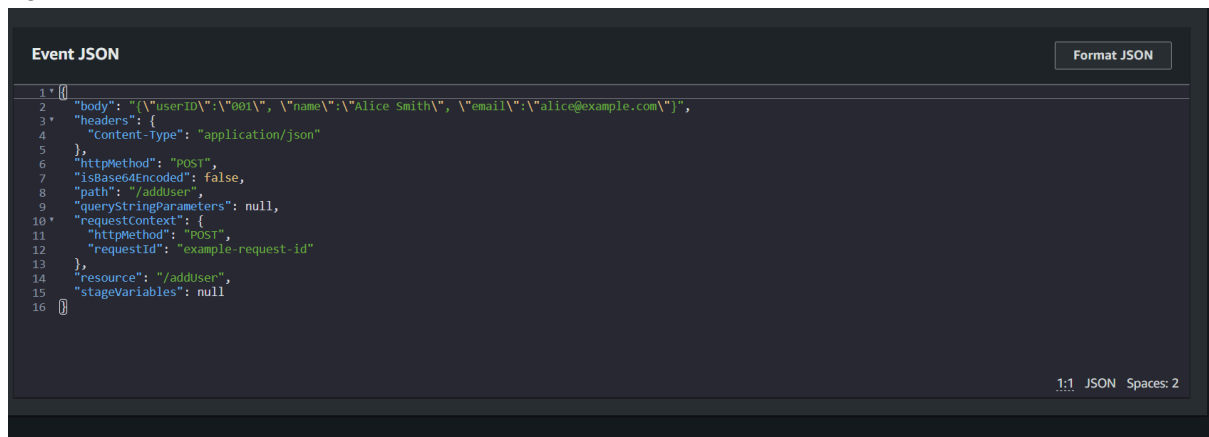
Navigate to the test section of the addUser function page and.

Paste this event json, in the event json code console, for testing the addUser lambda functions

Event json:

```
{
  "body": "{\"userID\":\"001\", \"name\":\"Alice Smith\", \"email\":\"alice@example.com\"}",
  "headers": {
    "Content-Type": "application/json"
  },
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "path": "/addUser",
  "queryStringParameters": null,
  "requestContext": {
    "httpMethod": "POST",
    "requestId": "example-request-id"
  },
  "resource": "/addUser",
  "stageVariables": null
}
```

Eg:



And the test runs fine here:

✓ Executing function: succeeded (logs 2)

Details

The area below shows the last 4 KB of the execution log.

```
{
  "statusCode": 200,
  "body": "{\\\"message\\\":\\\"User added successfully\\\",\\\"data\\\":{\\\"$metadata\\\":{\\\"httpStatusCode\\\":200,\\\"requestId\\\":\\\"07I088FDCIR5KR74J35TN00G4NVV4KQNS05AEMV3F66Q9ASUAAJG\\\",\\\"attempts\\\":1,\\\"totalRetryDelay\\\":0}}}"
}
```

Summary

Code SHA-256	Execution time
/9xKeeA4YU8Tk87R9f0kOoVP2mfG29qtZGP0vx2diw=	2 minutes ago
Request ID	Function version
089e2a8c-2f26-4124-9d4e-00efc72516c9	\$LATEST
Init duration	Duration
796.40 ms	889.42 ms

The addition is reflected in the table too.

RunReset

✓ Completed. Read capacity units consumed: 0.5

Items returned (1)

⌂ Actions Create item

< 1 > ⚙️

	UserID (String)	Email	Name
<input type="checkbox"/>	001	alice@exa...	Alice Smith

GetUser

Navigate to the test section of getUser function and add the event json

Eg:

```
{
  "queryStringParameters": {
    "userID": "001"
  },
  "httpMethod": "GET",
  "path": "/getUser",
  "headers": {
    "Content-Type": "application/json"
  },
  "requestContext": {
    "httpMethod": "GET",
    "requestId": "example-request-id",
    "path": "/getUser"
  },
  "resource": "/getUser",
  "stageVariables": null
}
```

Event name

getUserTest

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

Event sharing settings

☒ Private
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. [Learn more](#)

☐ Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. [Learn more](#)

Template - optional

hello-world

Event JSON Format JSON


```

1 {
2   "queryStringParameters": {
3     "userID": "001"
4   },
5   "httpMethod": "GET",
6   "path": "/getUser",
7   "headers": {
8     "Content-Type": "application/json"
9   },
10  "requestContext": {
11    "httpMethod": "GET",
12    "requestId": "example-request-id",
13    "path": "/getUser"
14  },
15  "resource": "/getUser",
16  "stageVariables": null
17 }
18

```

Do click on save though.

Test Results

 Executing function: succeeded ([logs](#) [2])

▼ Details

The area below shows the last 4 KB of the execution log.

```

{
  "statusCode": 200,
  "body": "{\"message\":\"User retrieved successfully\\\",\\\"data\\\":{\\\"userID\\\":\\\"001\\\",\\\"Name\\\":\\\"Alice Smith\\\",\\\"Email\\\":\\\"alice@example.com\\\"}}\""
}

```

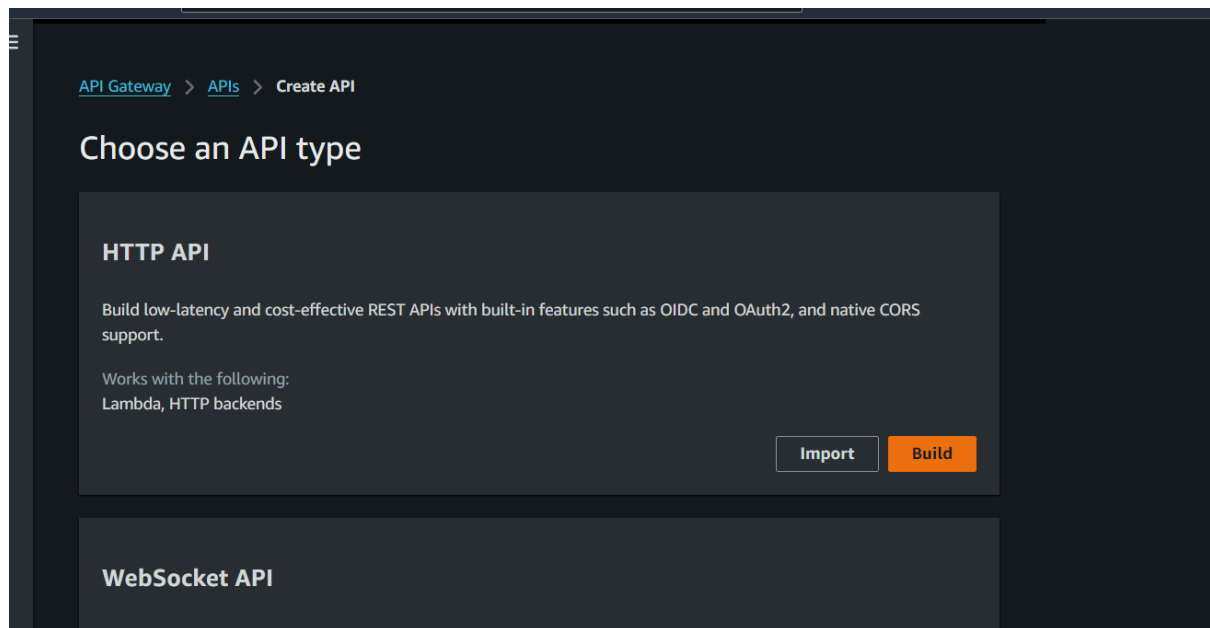
Summary

Code SHA-256	Execution time
q8E7Nexf5xxhKT9/d4bGpAYOXJYFAUJJ0UDj8OivK8E=	3 seconds ago
Request ID	Function version
2f308d9b-a5a6-490f-8775-66a601f51d27	\$LATEST
Init duration	Duration
337.88 ms	976.75 ms
Billed duration	Resources configured
977 ms	128 MB
Max memory used	
88 MB	

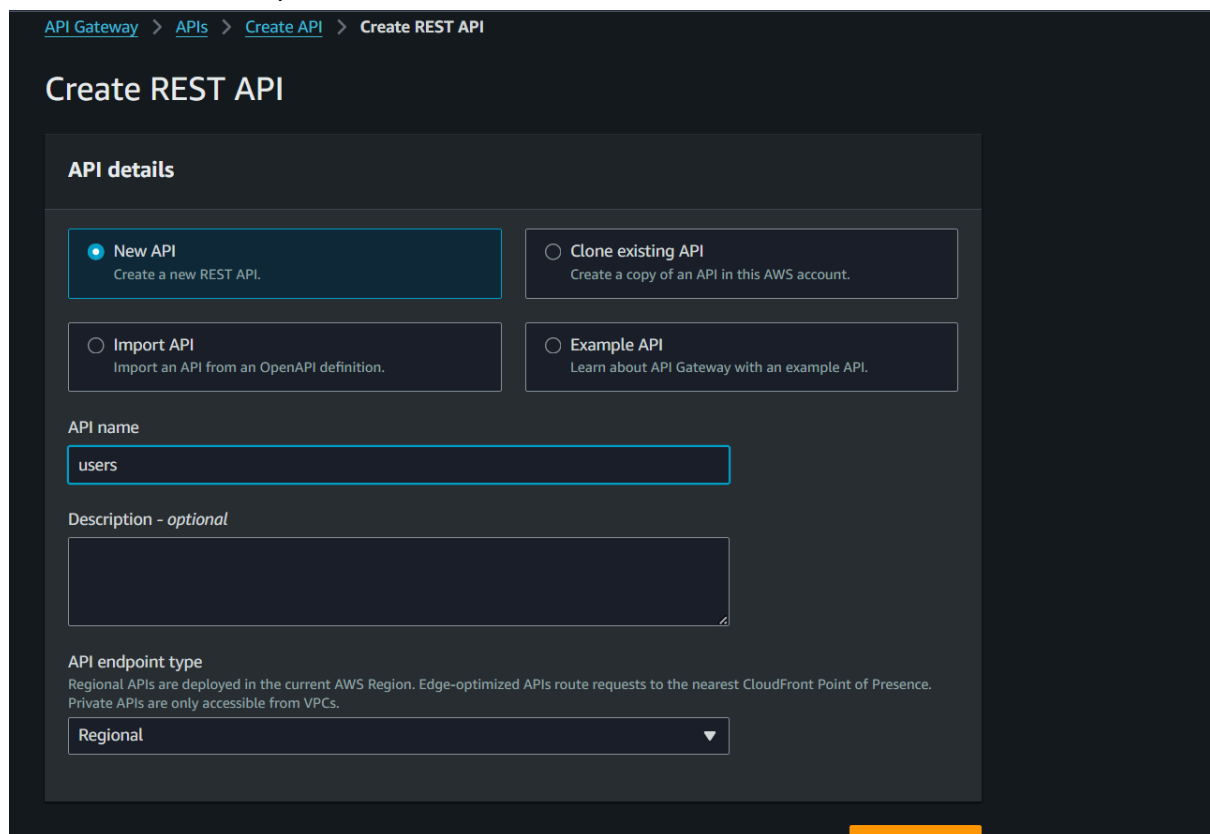
Works fine...

API gateway

Now navigate to the api gateway> APIs> Create API
Scroll down and select Rest API



Select the new API option and add a name to it. Click create



Post Method.

Click on create method

Method details


Method type

POST

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.




☐ AWS service

Integrate with an AWS Service.



☐ VPC link

Integrate with a resource that isn't accessible over the public internet.



☒ Lambda proxy integration

Send the request to your Lambda function as a structured event.

Lambda function

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

us-east-1

Q

arn:aws:lambda:us-east-1:434768569951:function:addl

X

Keeping everything else default:

Grant API Gateway permission to invoke your Lambda function. To turn off, update the function's resource policy yourself, or provide an invoke role that API Gateway uses to invoke your function.

Integration timeout

Info

By default, you can enter an integration timeout of 50 - 29,000 milliseconds. You can use Service Quotas to raise the integration timeout to greater than 29,000 ms

29000

Method request settings

Authorization

None

Request validator

None

☐ API key required

Operation name - optional

GetPets

URL query string parameters

No query strings found

Add query string

▼ HTTP request headers

No headers found

Add header

▼ Request body

No models found

Add model

CancelCreate method

✔ Successfully created method 'POST' in '/'. Redeploy your API for the update to take effect. X

API Gateway > APIs > Resources - users (iifv4r997d)

Resources

Create resource

/

POST

API actions ▼Deploy API

/ - POST - Method execution

Update documentationDelete

ARN

arn:aws:execute-api:us-east-1:434768569951:iifv4r997d/*/*POST/

Resource ID

s9jfw44pb0

Client

→

Method request

→

Integration request

→

Lambda integration

←

Integration response

←

Method response

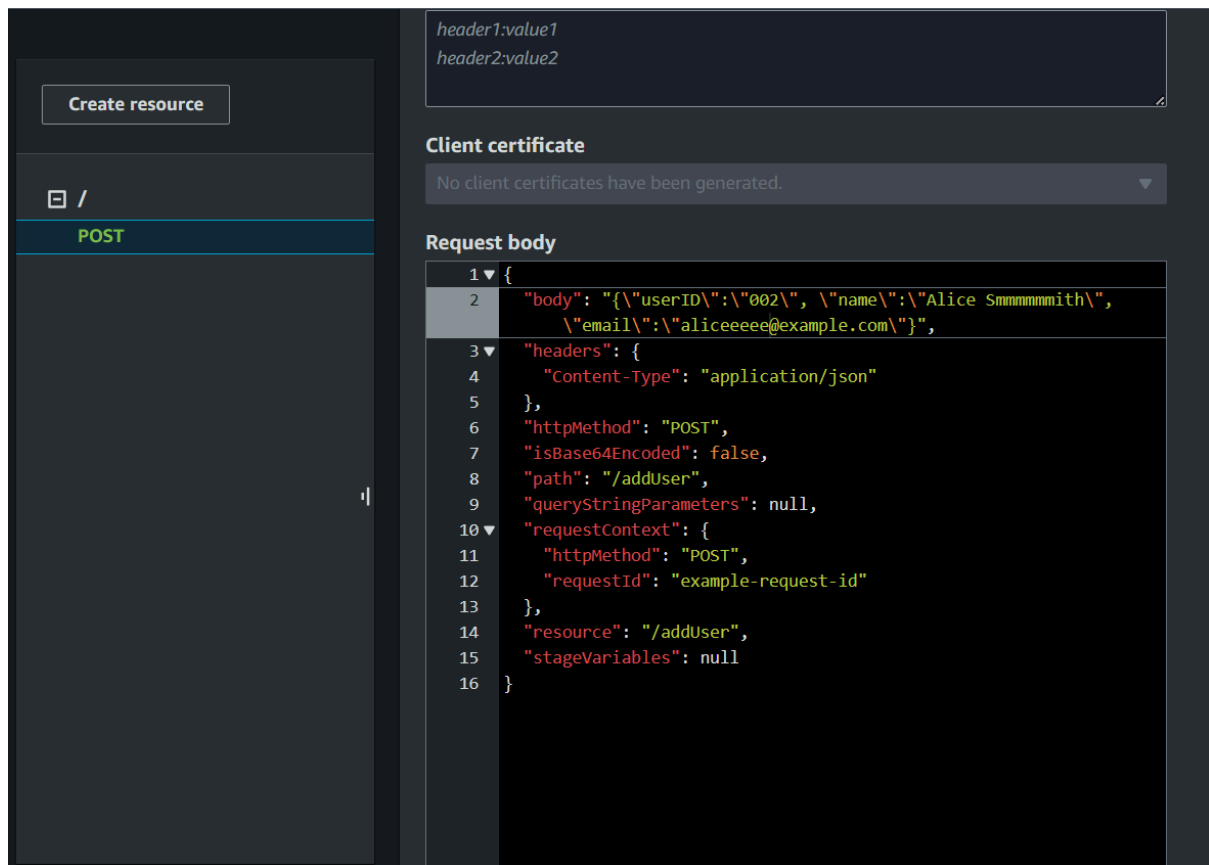
Method requestIntegration requestIntegration responseMethod responseTest

Method request settingsEdit

Click on Test

Add a new json (preferably in the same format as the test which you did.

Here's an example of what I did.



The code is :

```
{
  "body": "{\\"userID\\":\\"002\\", \\"name\\":\\"Alice Smmmmmmmith\\",
\\"email\\":\\"aliceeeee@example.com\\"}",
  "headers": {
    "Content-Type": "application/json"
  },
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "path": "/addUser",
  "queryStringParameters": null,
  "requestContext": {
    "httpMethod": "POST",
    "requestId": "example-request-id"
  },
  "resource": "/addUser",
  "stageVariables": null
}
```

The result of test:

The screenshot shows the test results for a POST method. It includes a table with Request, Latency ms, and Status. Below the table, it displays the Response body as a JSON object and the Response headers as a JSON object. The Logs section is also visible at the bottom.

Request	Latency ms	Status
/	1564	200

Response body

```
{
  "statusCode": 200,
  "body": {
    "message": "User added successfully",
    "data": {
      "httpStatusCode": 200,
      "requestId": "CLOANONDA68KKU482RN6TKEBNRVV4KQNSO5AEMVJF66Q9ASUAAJG",
      "attempts": 1,
      "totalRetryDelay": 0
    }
  }
}
```

Response headers

```
{
  "Content-Type": "application/json",
  "X-Amzn-Trace-Id": "Root=1-6714f7a7-d9b3fda1ad83d0175a6e4937;Parent=26a080cef8d1f620;Sampled=0;Lineage=1:0eba3a37:0"
}
```

Logs

Working fine...

Get Method

Now click on the / in the left pane again and again click on the create Method (in the right hand side)

The screenshot shows the AWS API Gateway console. At the top, there is a green notification bar stating 'Successfully created method 'POST' in '/'. Redeploy your API for the update to take effect.' Below this, the 'Resources' page is displayed. On the left, there is a sidebar with a tree view showing the resource hierarchy: API Gateway > APIs > Resources - users (iifv4r997d). The main area shows the 'Resources' section with a 'Create resource' button. Below this, there is a table of resources. The resource '/' is selected, and its details are shown on the right. The 'Methods (1)' section shows a table with one method: POST, using Lambda integration, with no authorization and no API key required.

API Gateway > APIs > Resources - users (iifv4r997d)

Resources

API actions ▼ Deploy API

Create resource

Path	Resource ID
/	s9jfw44pb0

Methods (1) Delete Create method

Method type	Integration type	Authorization	API key
POST	Lambda	None	Not required






Now select GET as the method type, choose lambda.

Create method

Method details

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.

- ☐ **AWS service**
Integrate with an AWS Service.

- ☐ **VPC link**
Integrate with a resource that isn't accessible over the public internet.


☐ **Lambda proxy integration**
Send the request to your Lambda function as a structured event.

Lambda function

Select the getUser lambda function, and click on the create method, keeping everything else as it is.

Send the request to your Lambda function as a structured event

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

us-east-1

arn:aws:lambda:us-east-1:434768569951:function:getUser

☒ **Grant API Gateway** ☐ **Grant API Gateway** ☐ **Grant API Gateway**

Integration timeout [Info](#)
By default, you can enter an integration timeout of 50 - 29,000 milliseconds. You can use Service Quotas to raise the integration timeout to greater than 29,000 ms

29000

► **Method request settings**

► **URL query string parameters**

► **HTTP request headers**

► **Request body**

Cancel Create method

Now Before testing one important step is there:
Go to the integration request section...

The screenshot shows the AWS API Gateway console interface for a specific method. On the left, there's a sidebar with a 'Create resource' button and a list of methods including 'GET' and 'POST'. The main area is titled '/ - GET - Method execution' and includes buttons for 'Update documentation' and 'Delete'. It displays the ARN 'arn:aws:execute-api:us-east-1:434768569951:iifv4r997d/*/GET/' and the Resource ID 's9jfw44pb0'. A diagram shows the flow: Client → Method request → Integration request → Lambda integration. Below this, there are tabs for 'Method request', 'Integration request' (which is selected), 'Integration response', 'Method response', and 'Test'. The 'Integration request settings' section shows 'Integration type' as 'Lambda' and 'Region' as 'us-east-1'. There is an 'Edit' button next to the settings.

Click on edit.

Scroll down> click on mapping templates> then click on add mapping template

Write "application/json" in content type.

And write this code in the Template body:

```
{
  "queryStringParameters": {
    "userID": "$input.params('UserID')"
  }
}
```

▼ Mapping templates

Content type

application/json

Remove

Generate template

Template body

1 {

2 "queryStringParameters": {

3 "userID": "\$input.params('UserID')"

4 }

5 }

6 }

Click on save.

Now navigate to the test section of this GetMethod and paste the query string here:

Eg:

My query string :

UserID=001

GET
POST

Method request

Integration request

Integration response

Method response

Test

Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

UserID=001

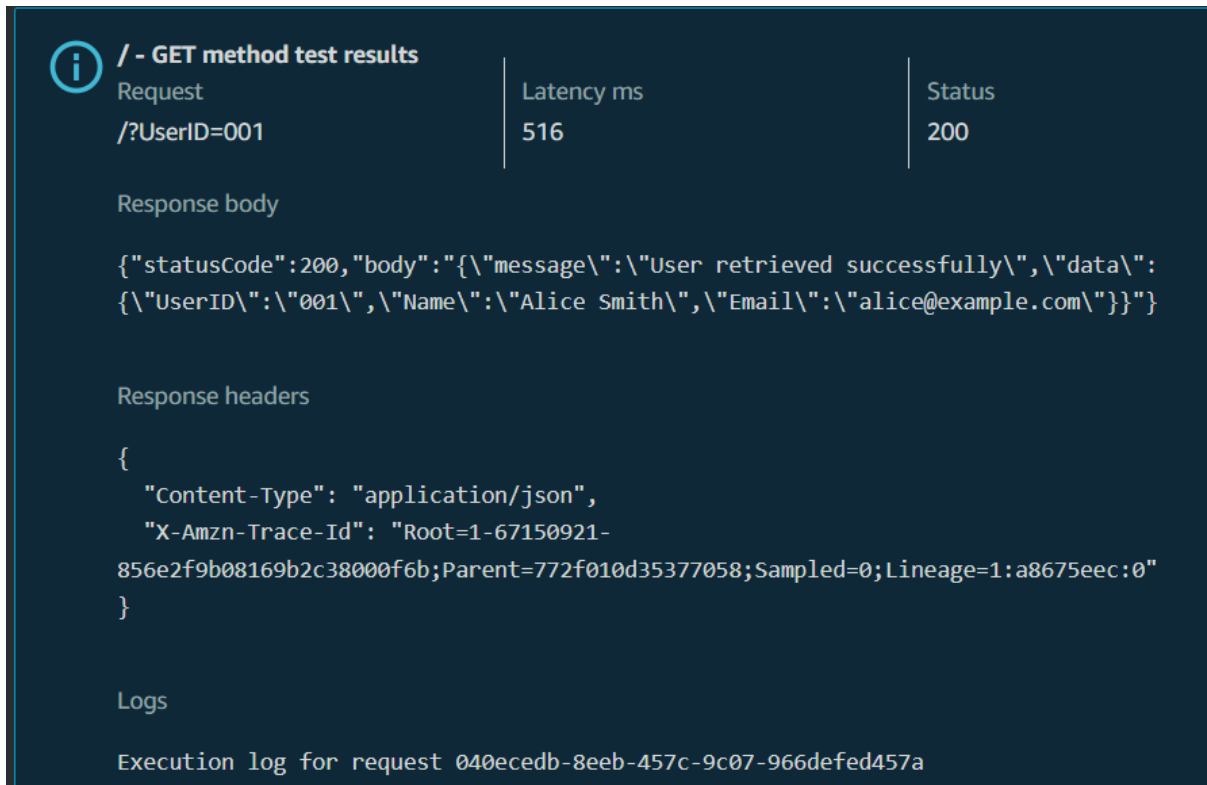
Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

Results:



The screenshot shows the results of a GET request in Postman. It includes a table for request details, the response body in JSON, response headers, and a log entry.

/ - GET method test results		
Request	Latency ms	Status
/?UserID=001	516	200

Response body

```
{"statusCode":200,"body":{"message":"User retrieved successfully","data":{"UserID":"001","Name":"Alice Smith","Email":"alice@example.com"}}
```

Response headers

```
{  "Content-Type": "application/json",  "X-Amzn-Trace-Id": "Root=1-67150921-856e2f9b08169b2c38000f6b;Parent=772f010d35377058;Sampled=0;Lineage=1:a8675eec:0"}
```

Logs

Execution log for request 040ecedb-8eeb-457c-9c07-966defed457a

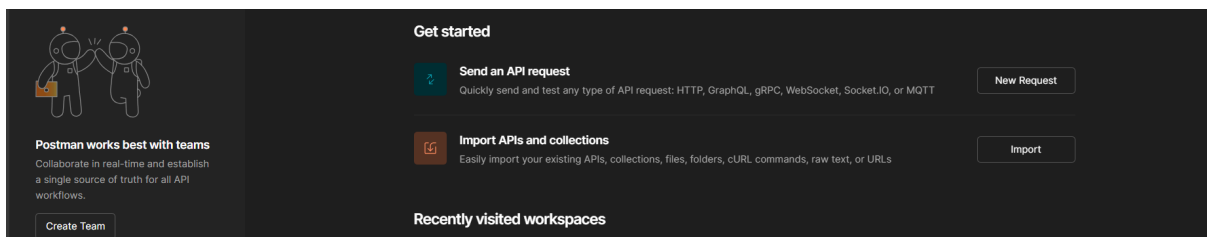
works fine...

Click on deploy..

Testing the API (Using postman web)

Open postman web sign in with your account.

Click on send an API request

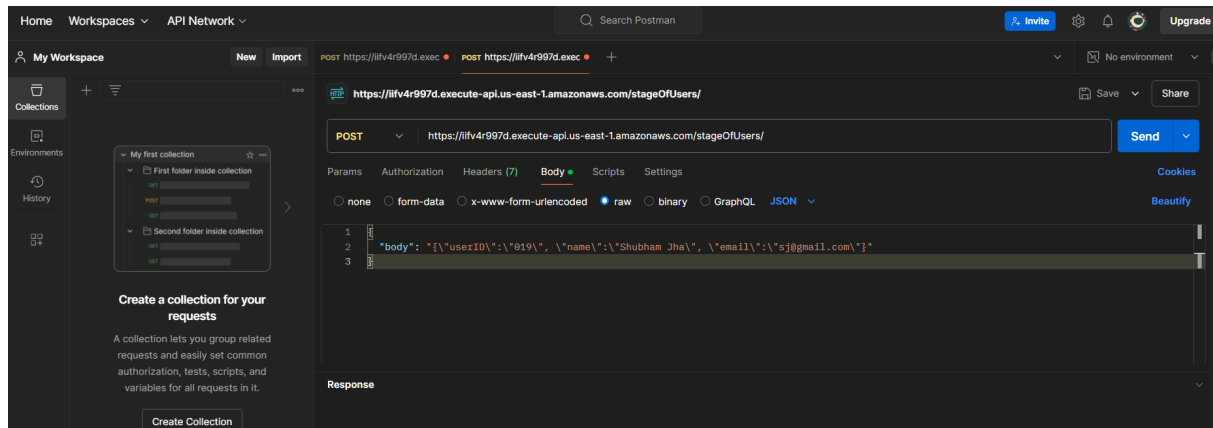


The screenshot shows the Postman web interface. It includes a sidebar with a 'Create Team' button, a main area with 'Get started' options like 'Send an API request' and 'Import APIs and collections', and a 'Recently visited workspaces' section.

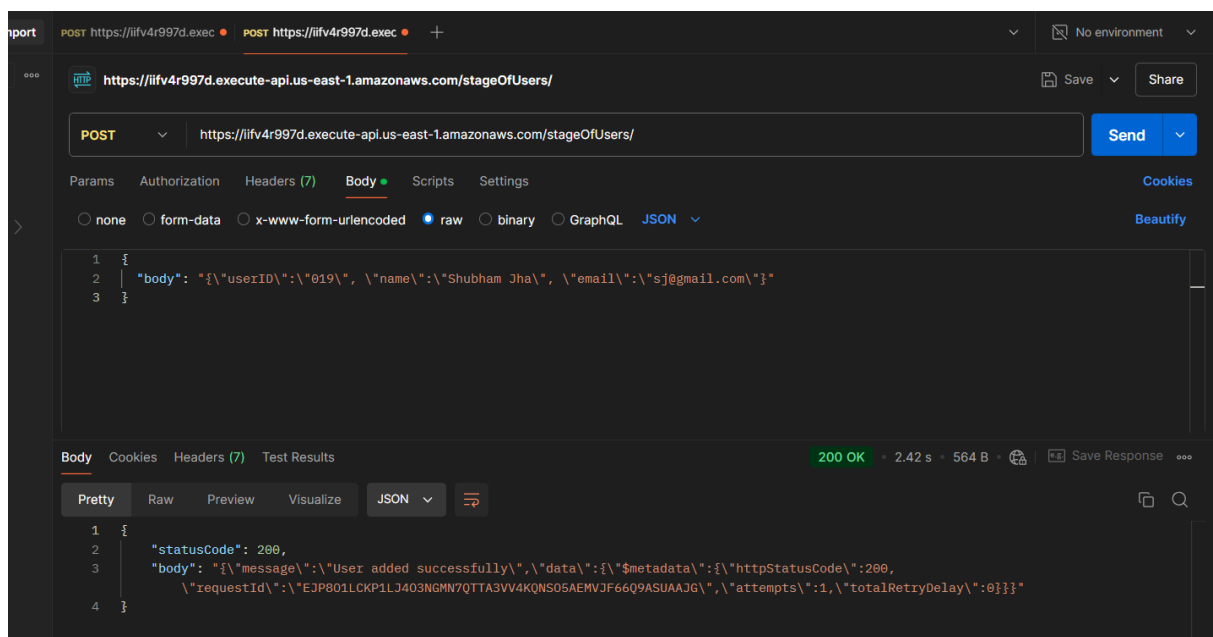
Paste the API link that you got after clicking deploy.

Then click on body, select raw and paste the body part of the test that you wrote (event Json) in the addUser Lambda function.

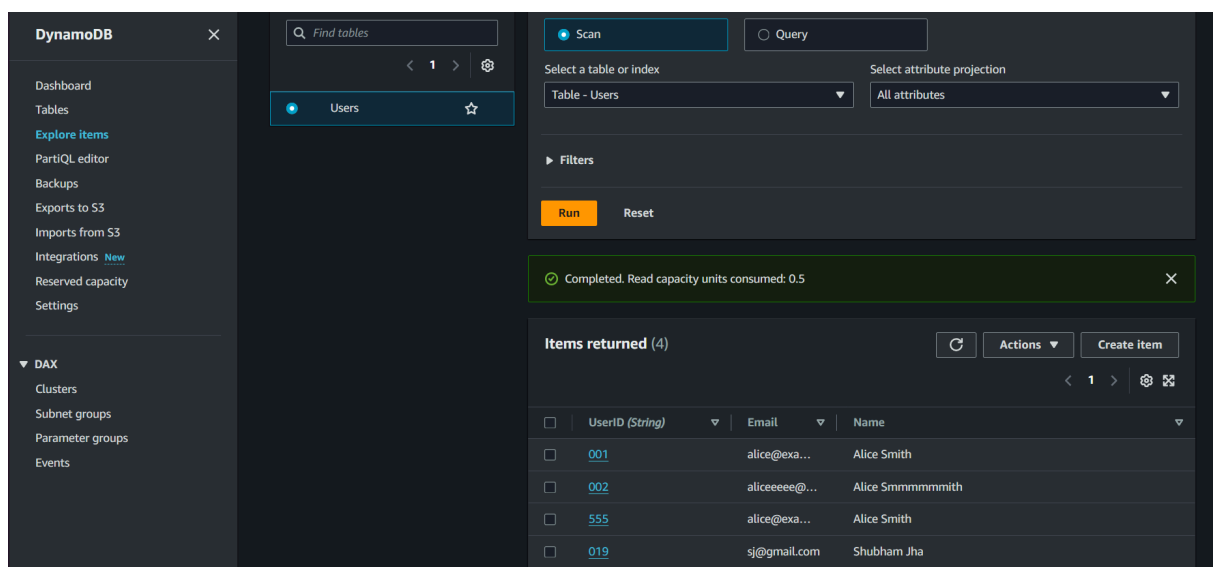
Eg:



Click on send,



Check the table:

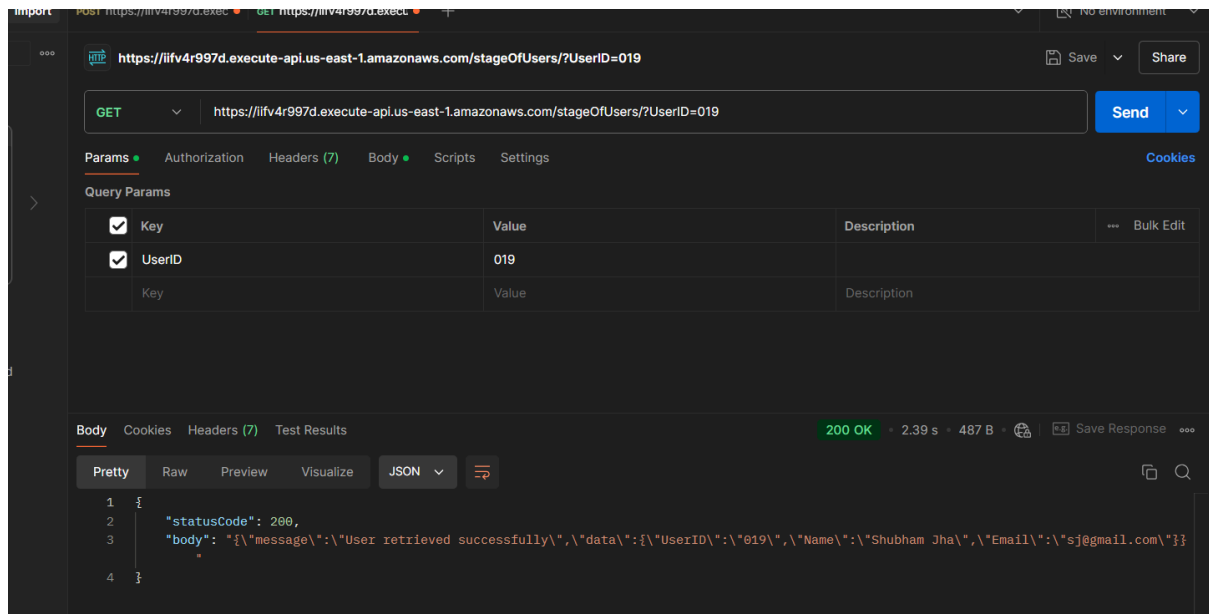


Working fine...

Check get method:

Change the method to get:

Add the parameters in the params section.



Conclusion:

In conclusion, building a serverless REST API with AWS Lambda, API Gateway, and DynamoDB is a smart and efficient way to develop modern applications. AWS Lambda handles the heavy lifting by running your code without worrying about servers, while API Gateway smoothly directs HTTP requests to the right Lambda functions. DynamoDB offers fast, reliable storage for user data, perfectly fitting into this setup. Together, they simplify the development process while providing strong performance, easy scaling, and top-notch security—making this approach ideal for managing user data in web apps.