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.

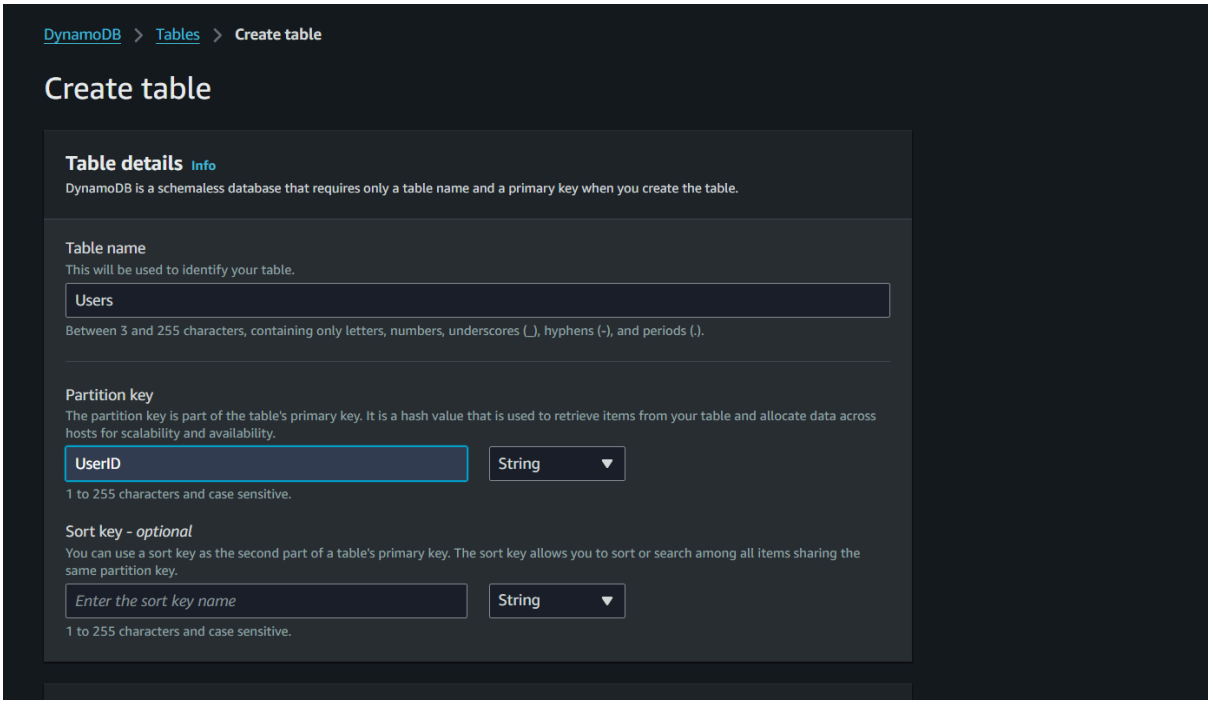
3rd year Project Relation

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.

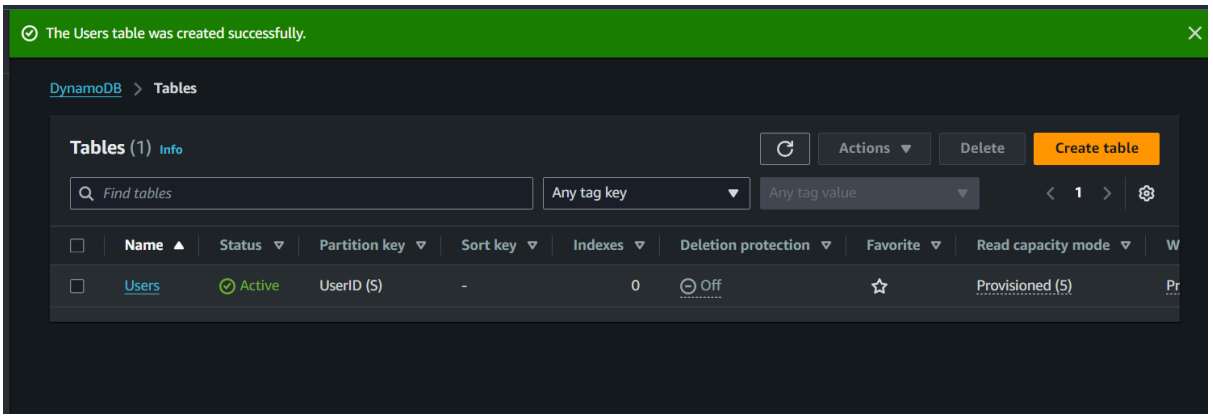
Step by Step Execution:

Dynamo DB:

Navigate to DynamoDB dashboard>Tables>Click on create table.
I have named the table as Users
Partition key of Users = UserID (string)



Click on create table and you will see this:

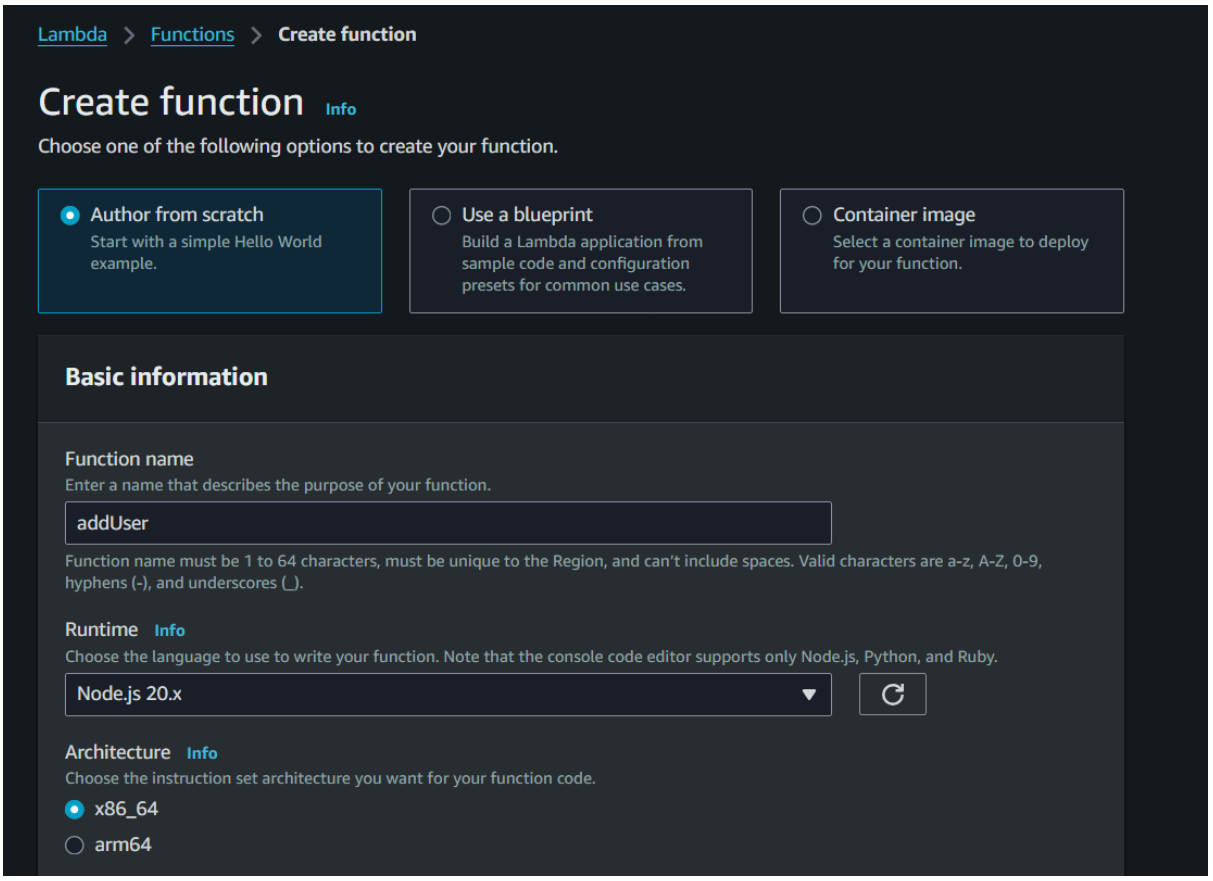


Lambda Functions

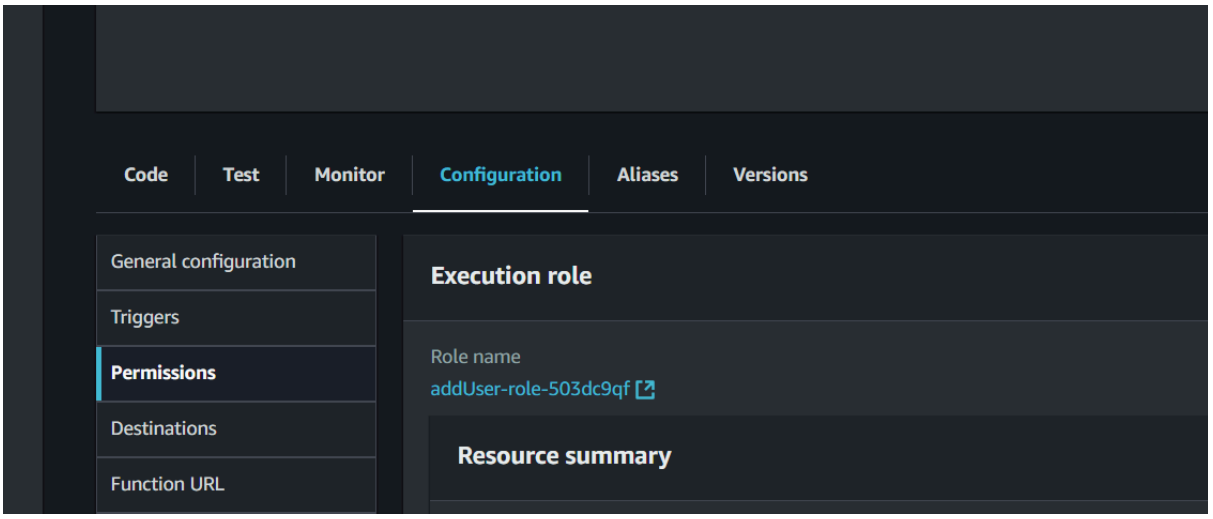
Navigate to Lambda dashboard.

AddUserFunction

Click on create function and the following details.



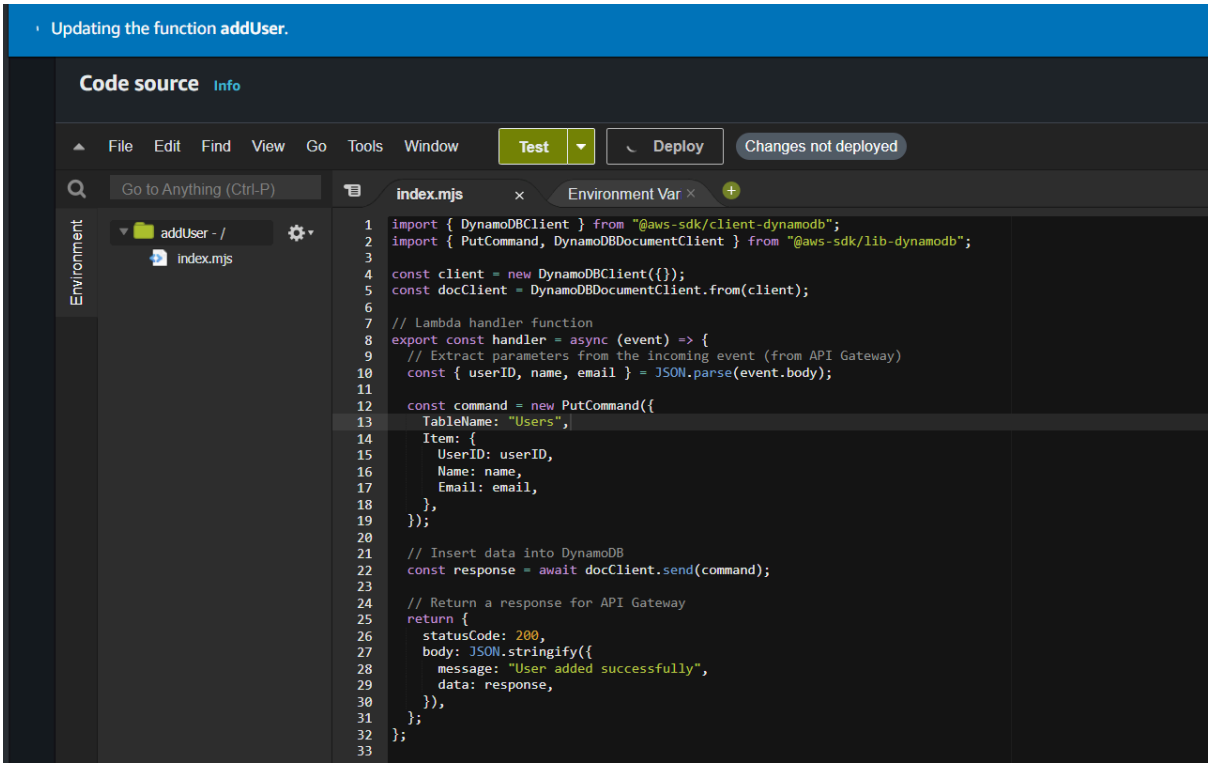
Keep everything default and click on create function.
Navigate to configuration, then select Permissions to note the execution role name



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:

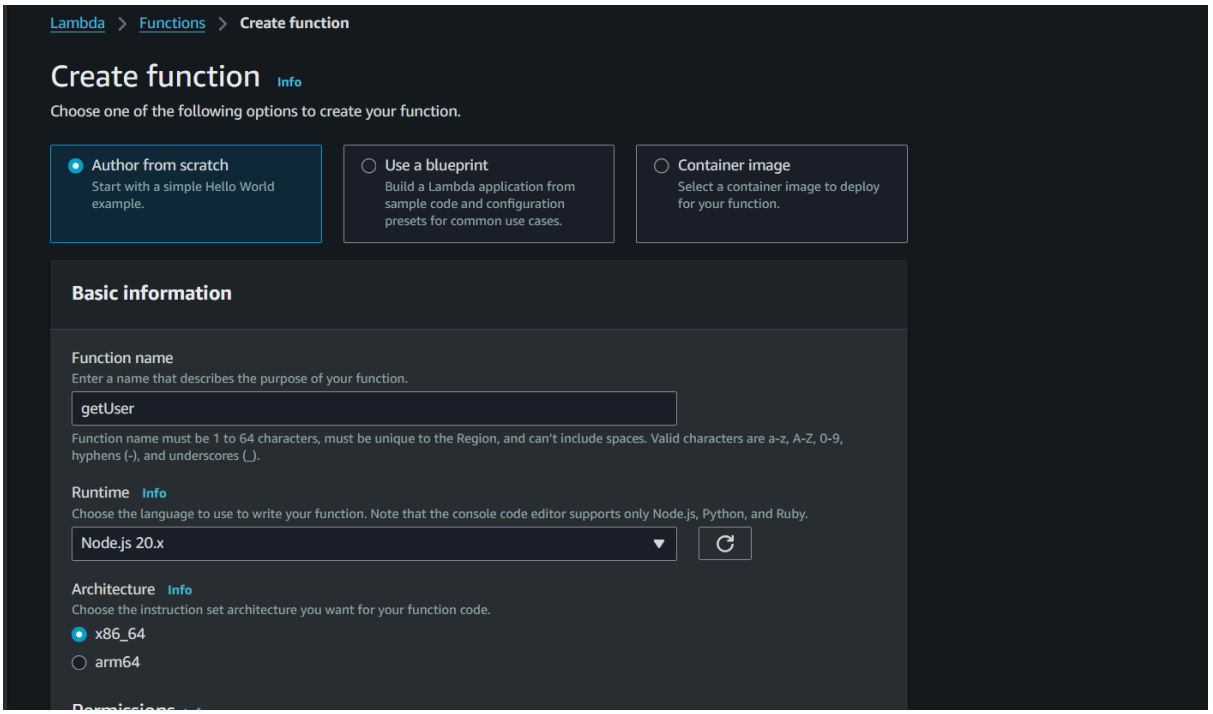
```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

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,
    }),
  };
};
```



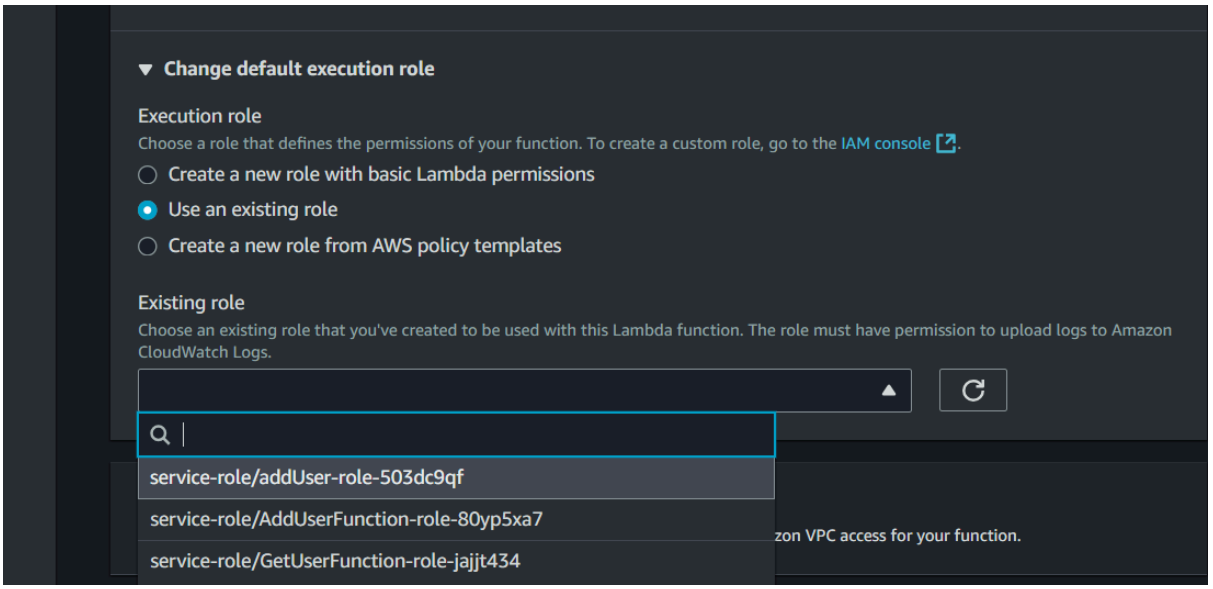
The tests are done in the IAM role config test section.

GetUserFunction

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



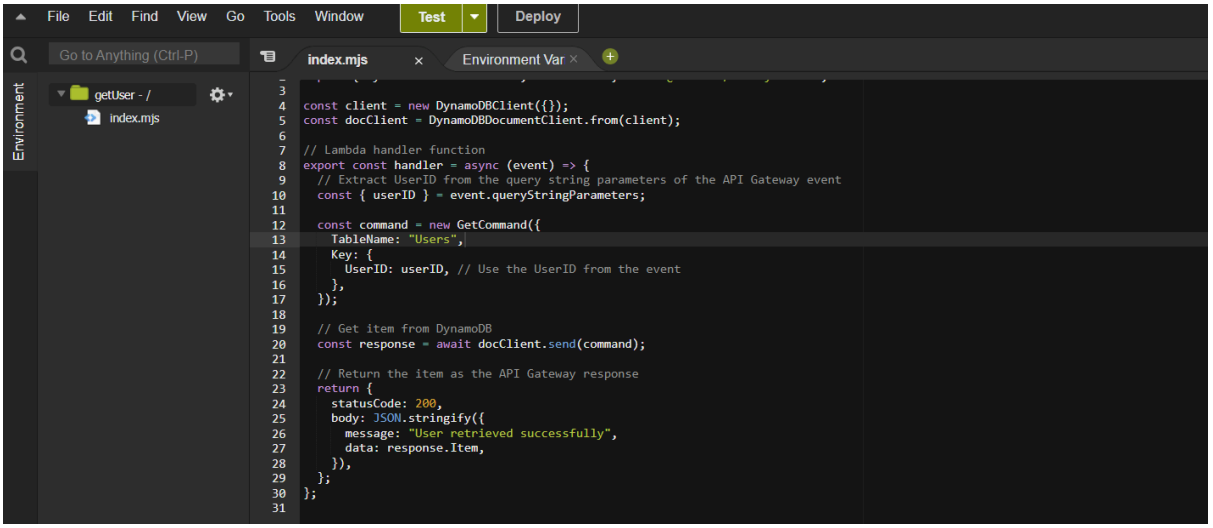
Keep everything default except this time select an existing role which we noted earlier.



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({
```

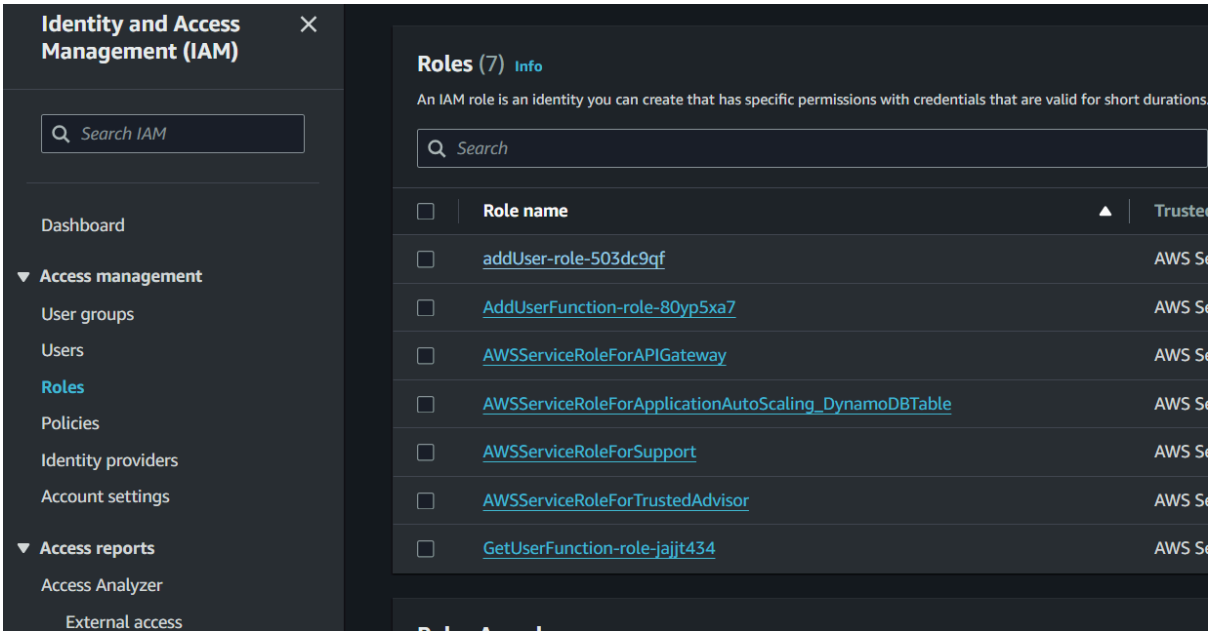
```
      message: "User retrieved successfully",
      data: response.Item,
    }),
  };
};
```



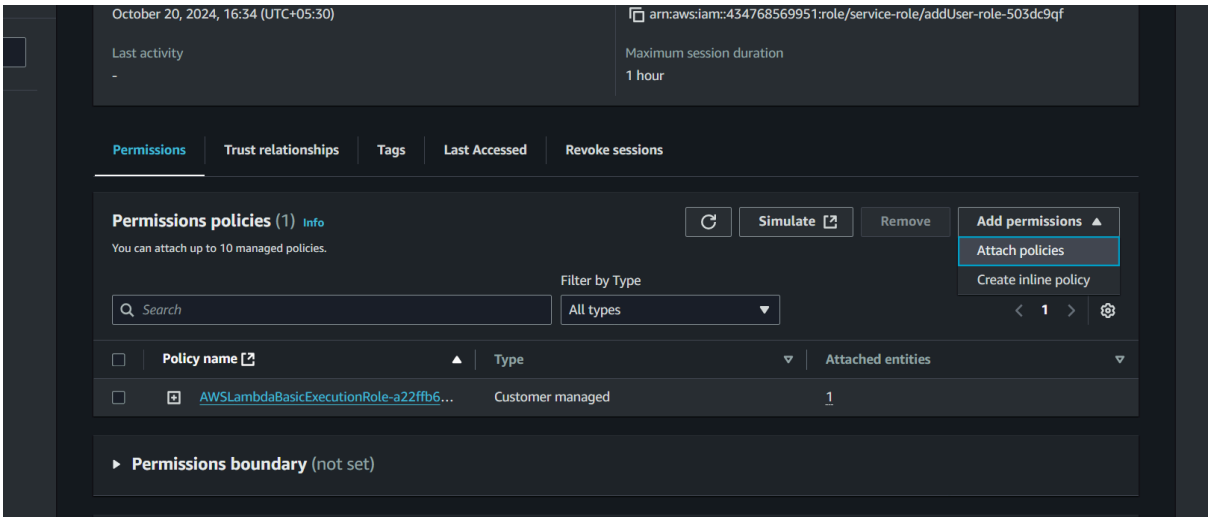
The testing is done in the Test Lambda section in the IAM role config.

IAM role config

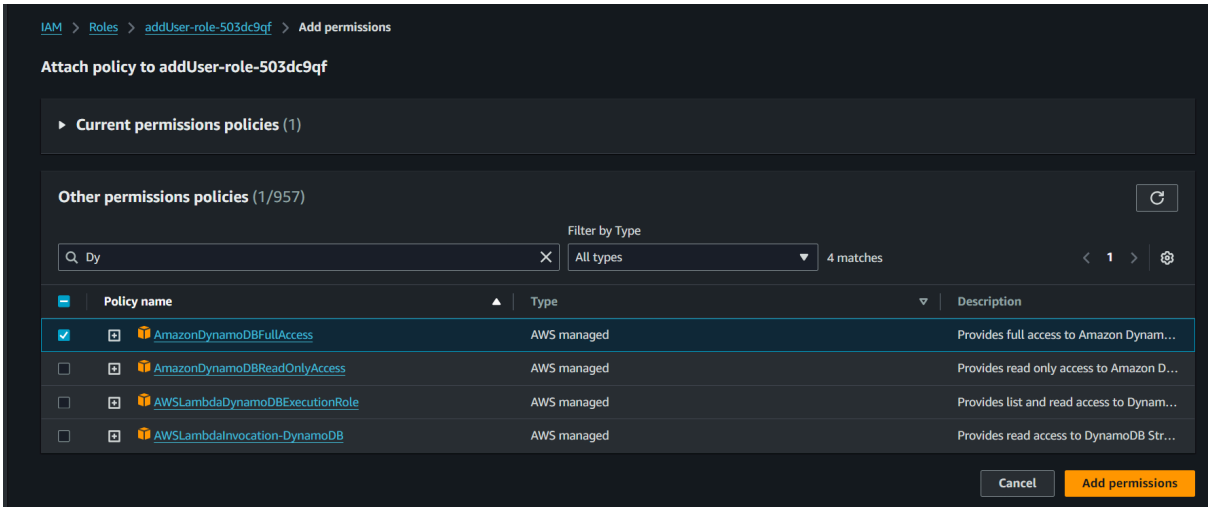
Navigate to IAM roles. Find the role which got created automatically when creating the adduser function and used in both of the lambda functions.
Click on that role:



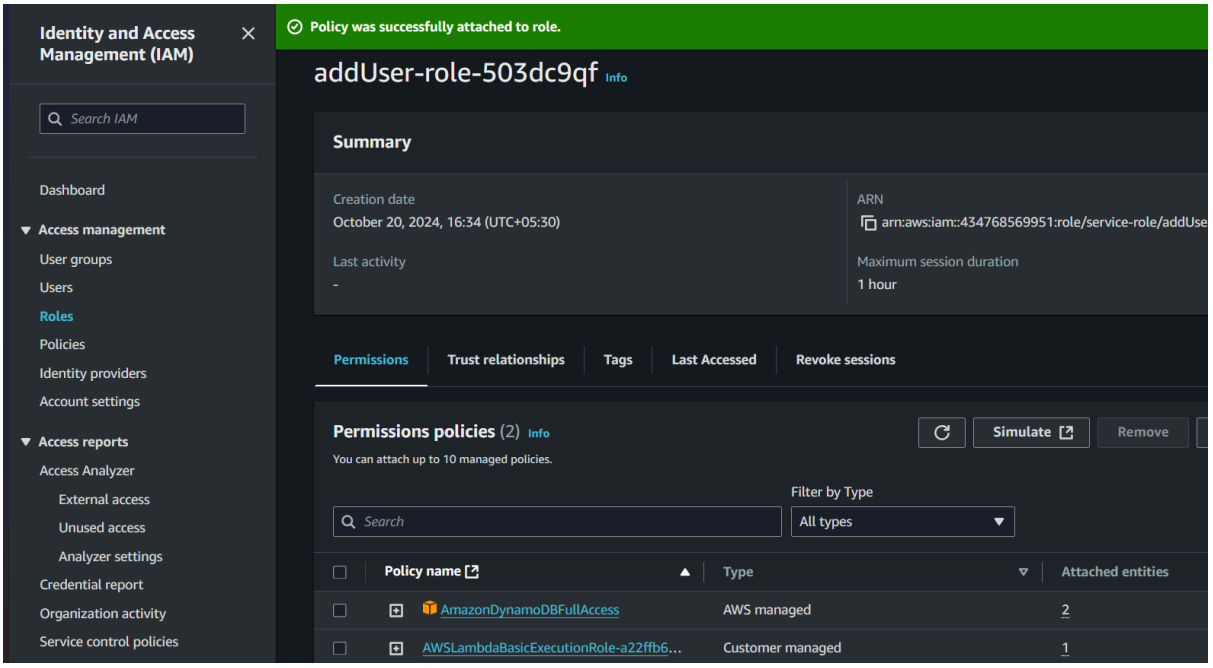
Click on add permissions>attach policies



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

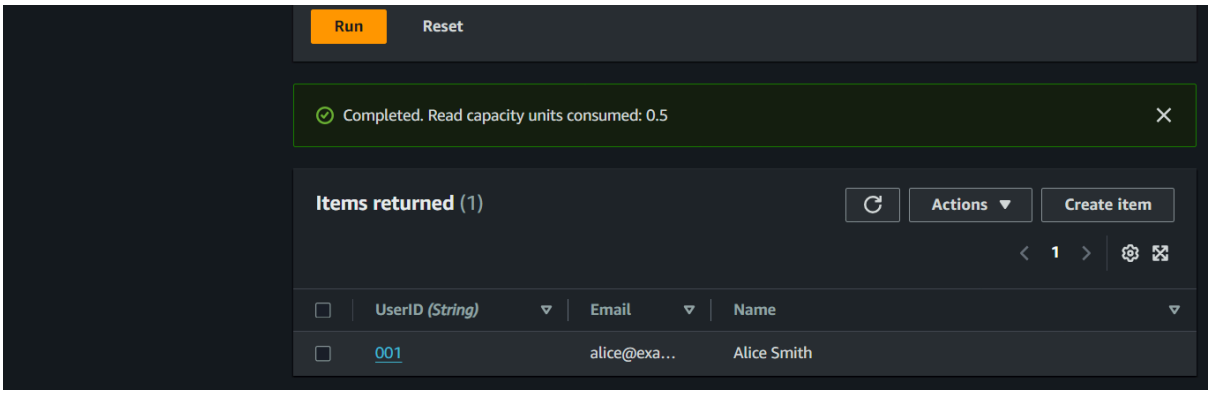
Event json:


```
{
  "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:

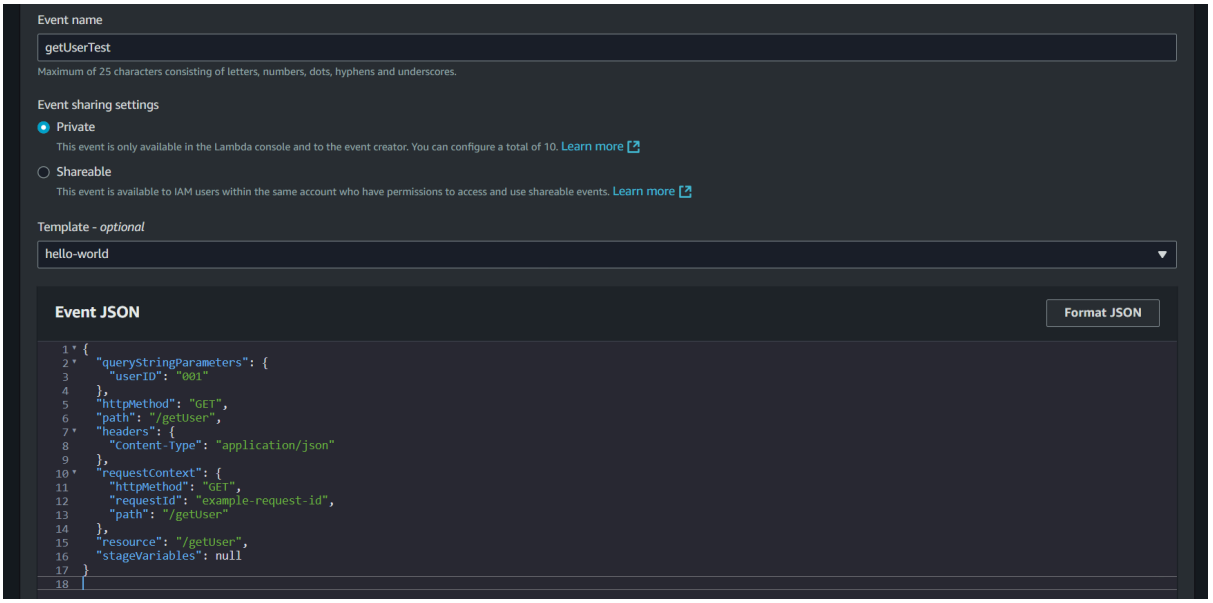
The addition is reflected in the table too.



GetUser

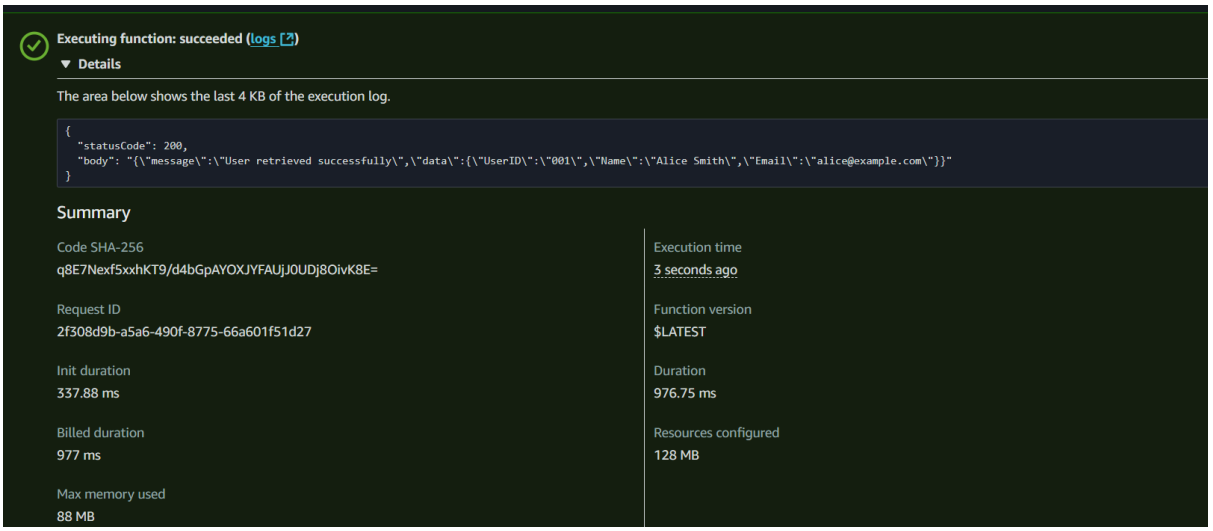
Navigate to the test section of getUser function and add the event json
Eg:

```
{
  "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
}
```



Do click on save though.

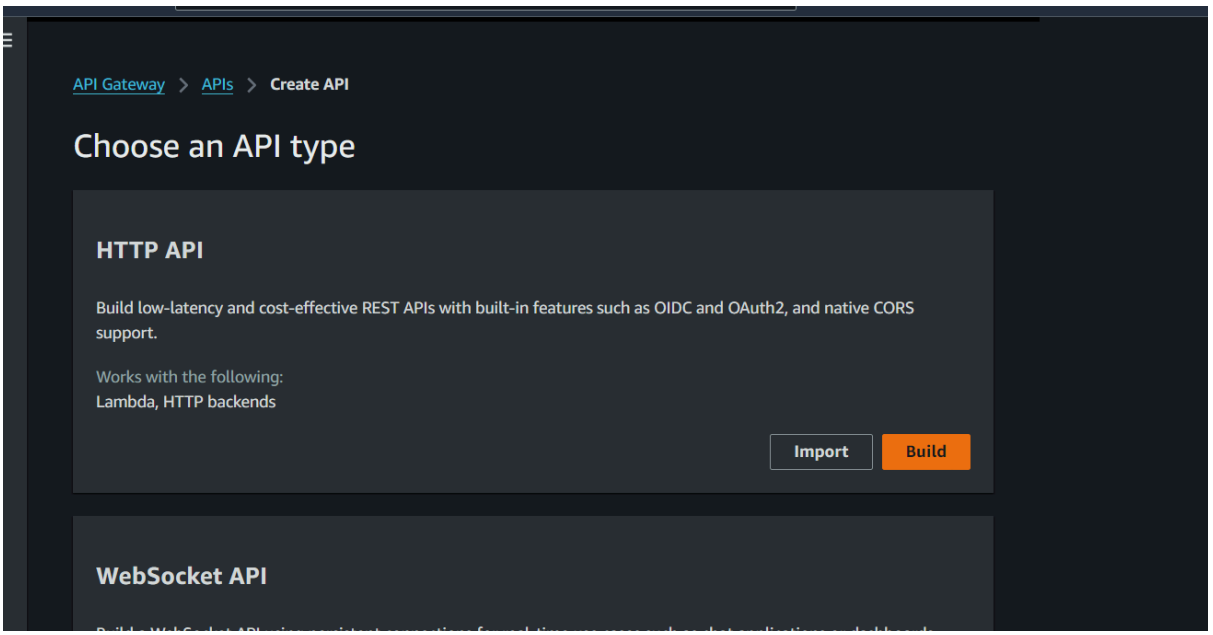
Test Results



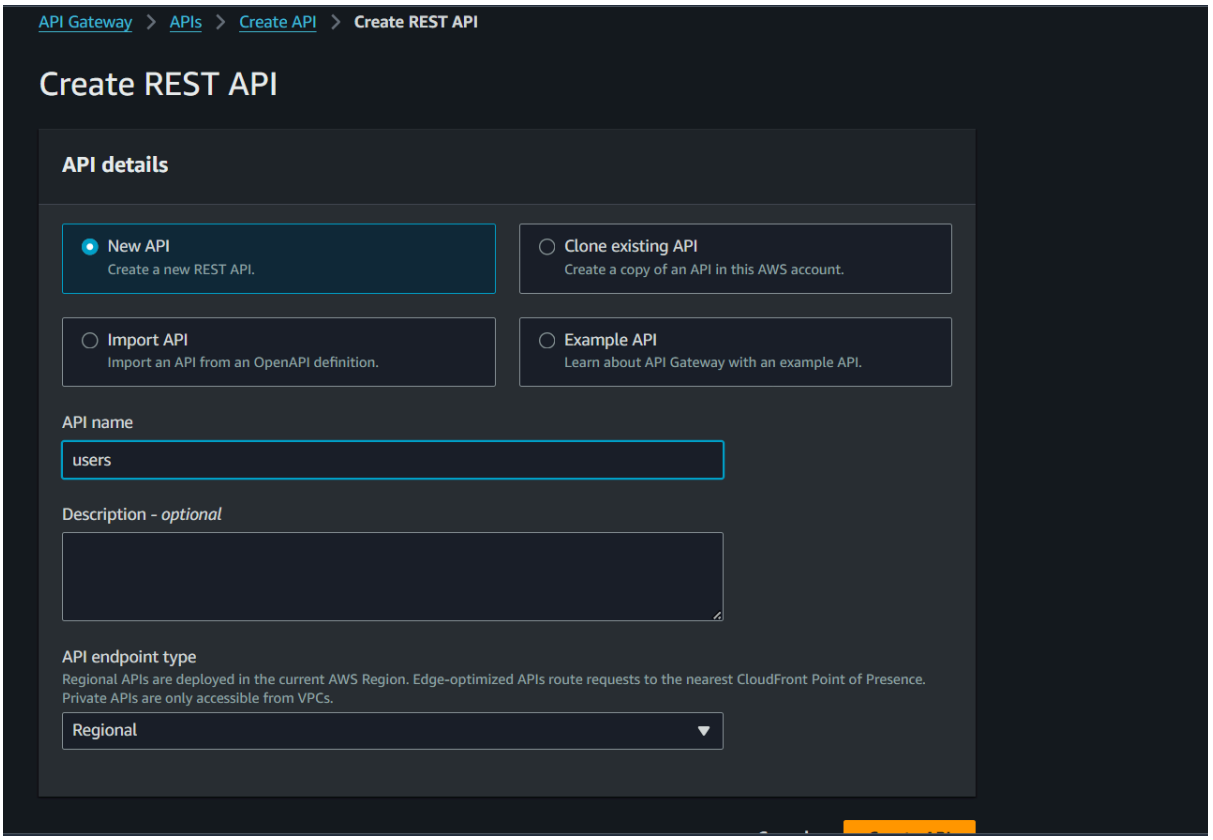
Works fine…

API gateway

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

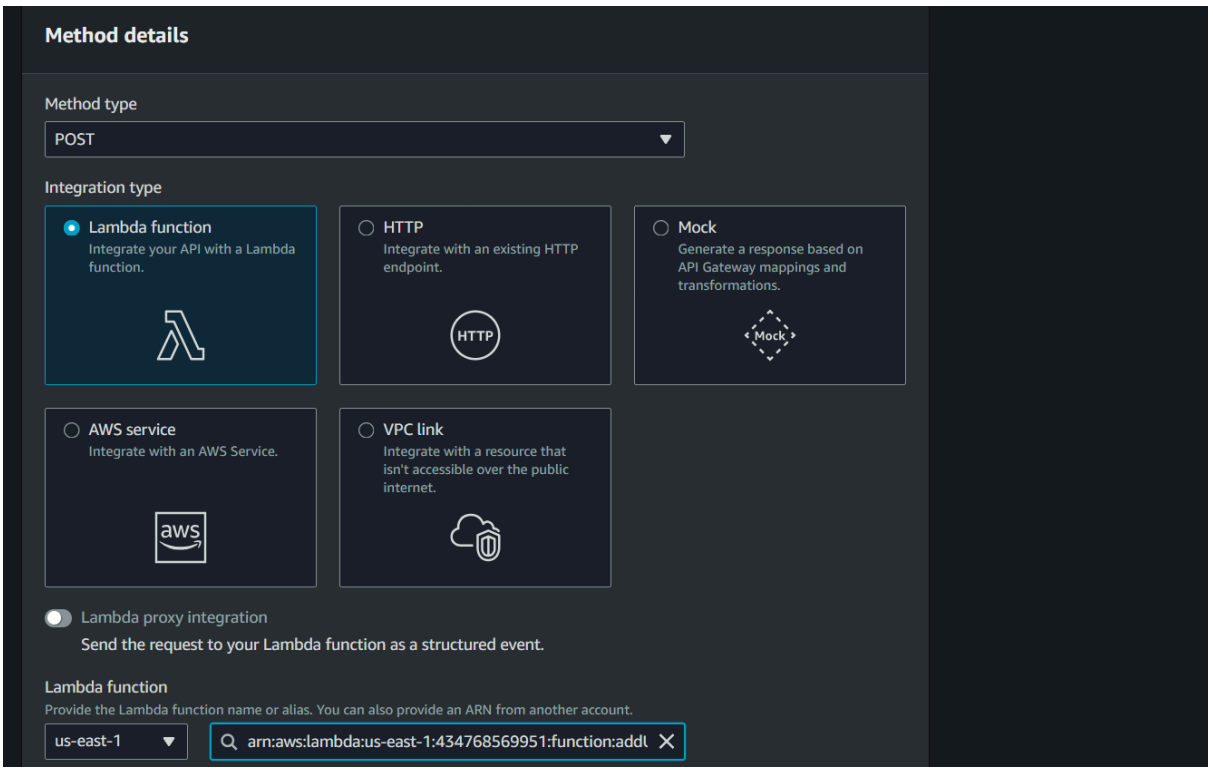


Select the new API option and add a name to it. Click create

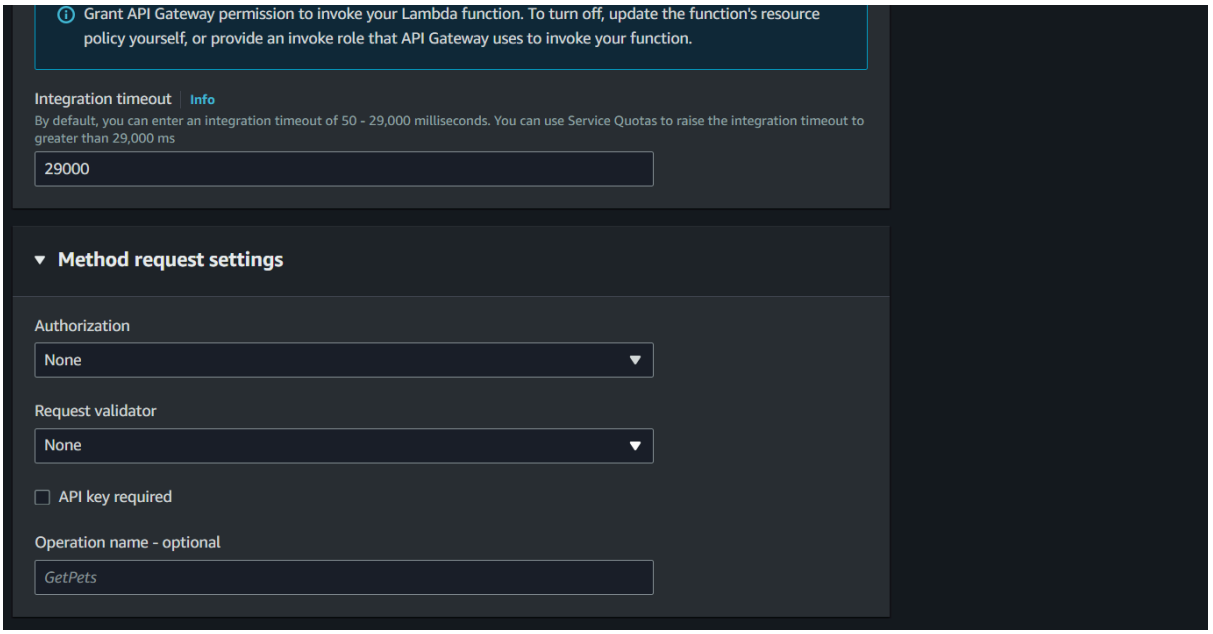


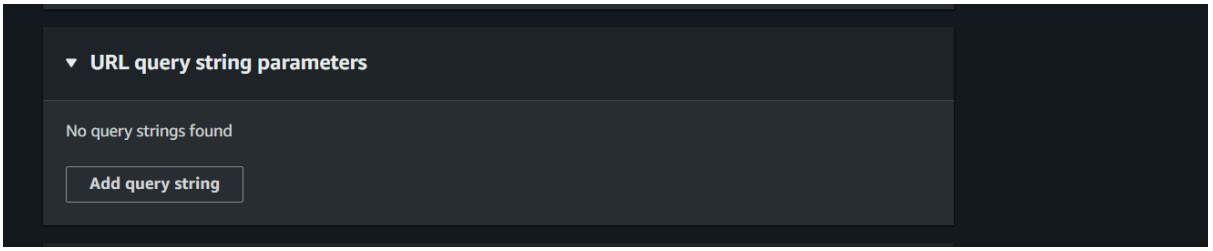
Post Method.

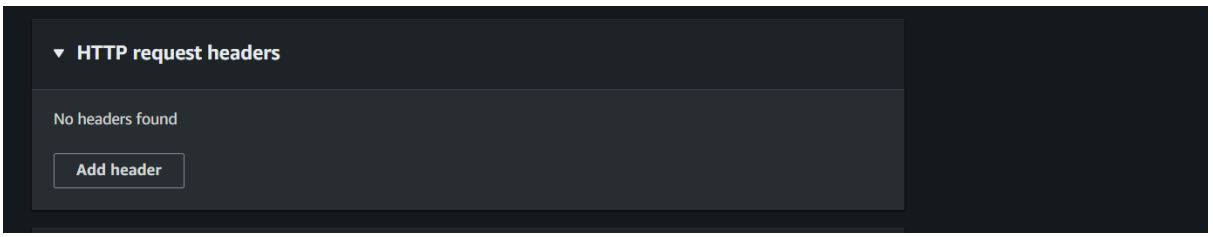
Click on create method

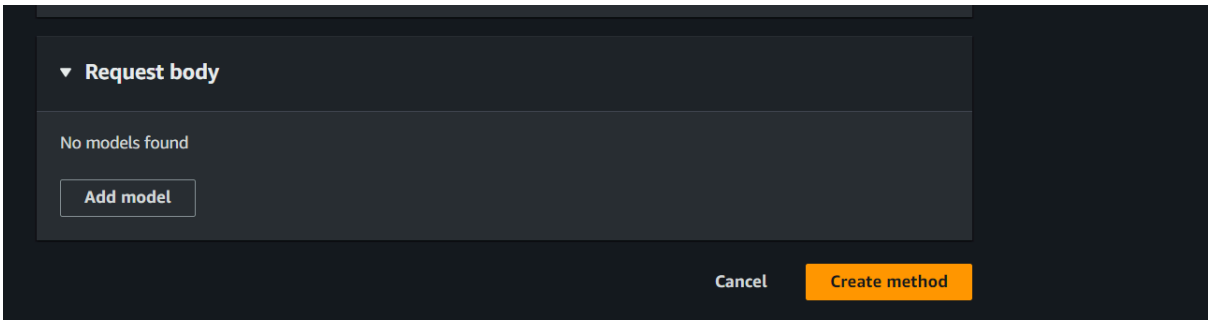


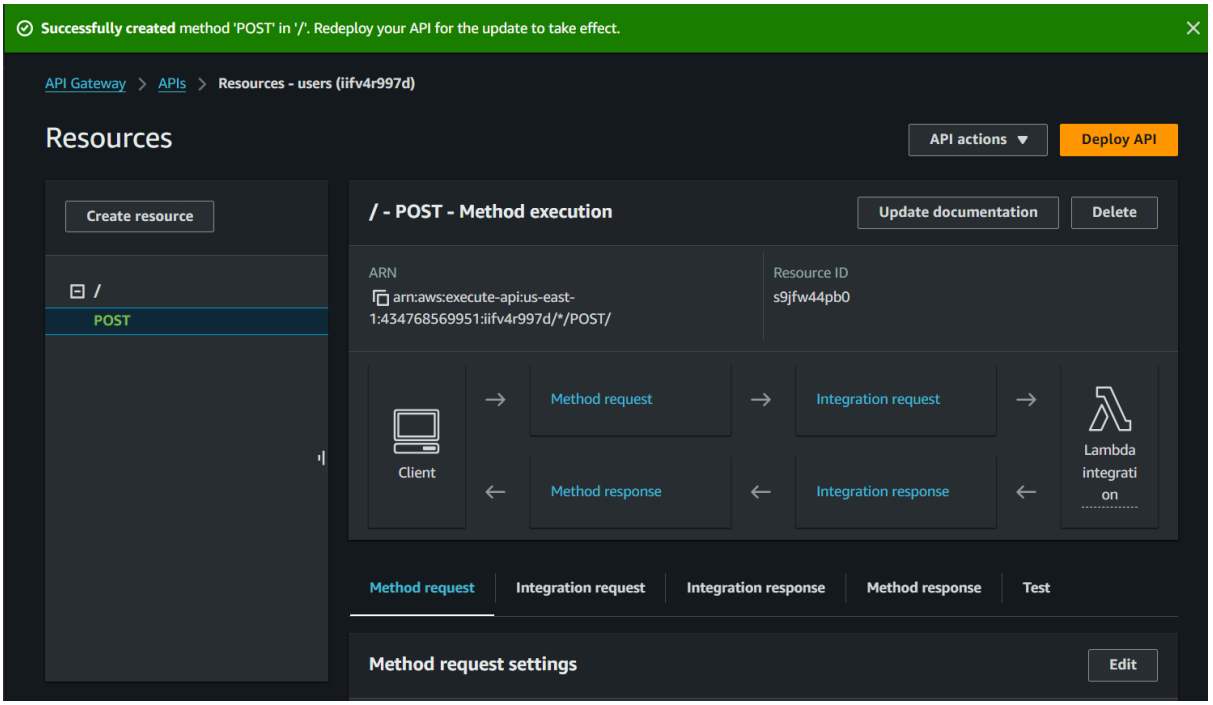
Keeping everything else default:







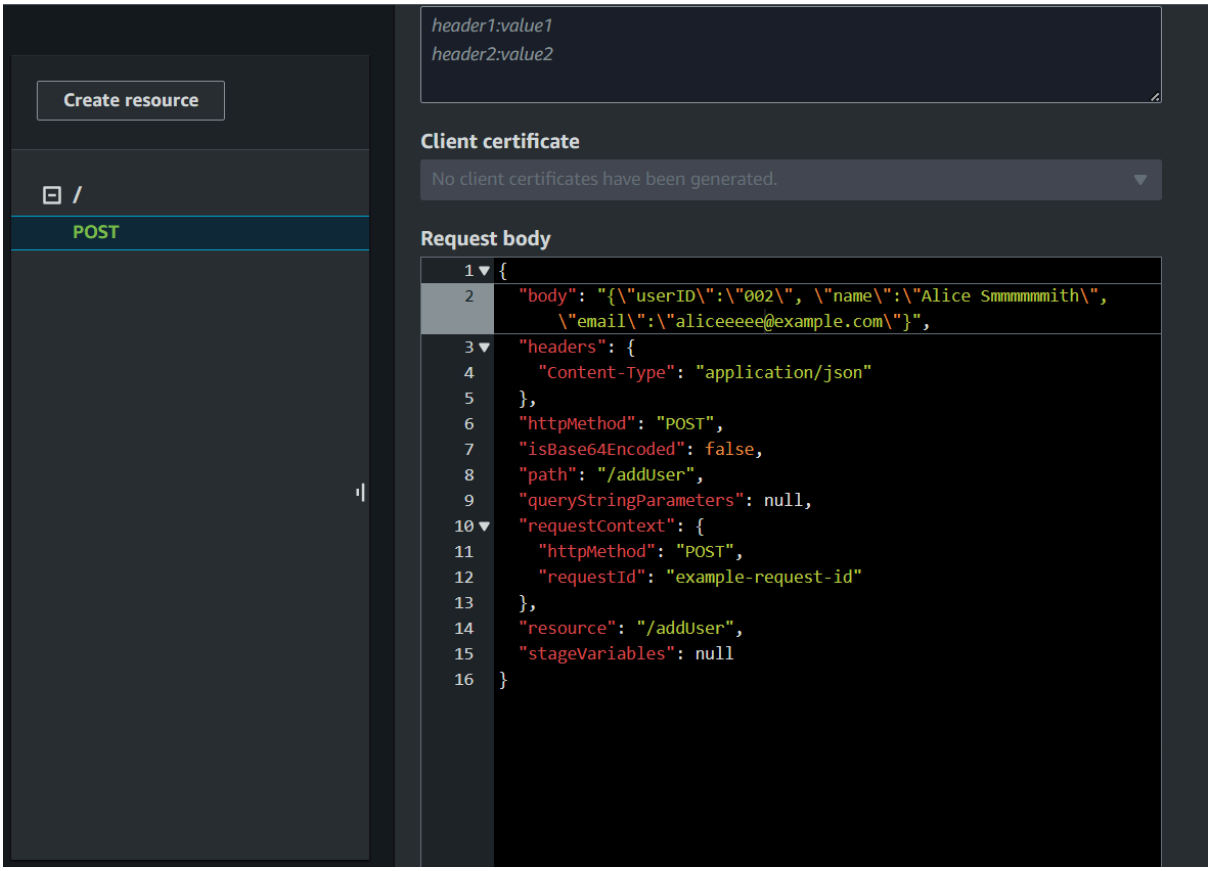




Click on Test

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 :

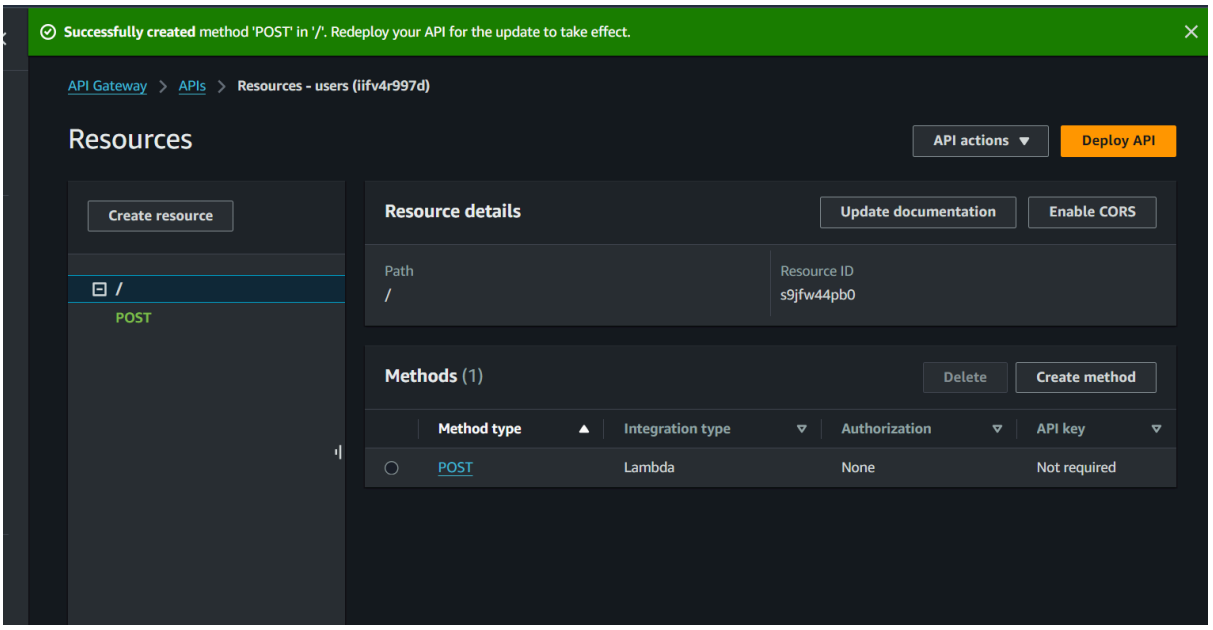

```
{
  "body": "{\"userID\":\"002\", \"name\":\"Alice Smmmmmmith\",
\"email\":\"aliceeeee@example.com\"}",
  "headers": {
    "Content-Type": "application/json"
  },
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "path": "/addUser",
  "queryStringParameters": null,
  "requestContext": {
    "httpMethod":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

The result of test:

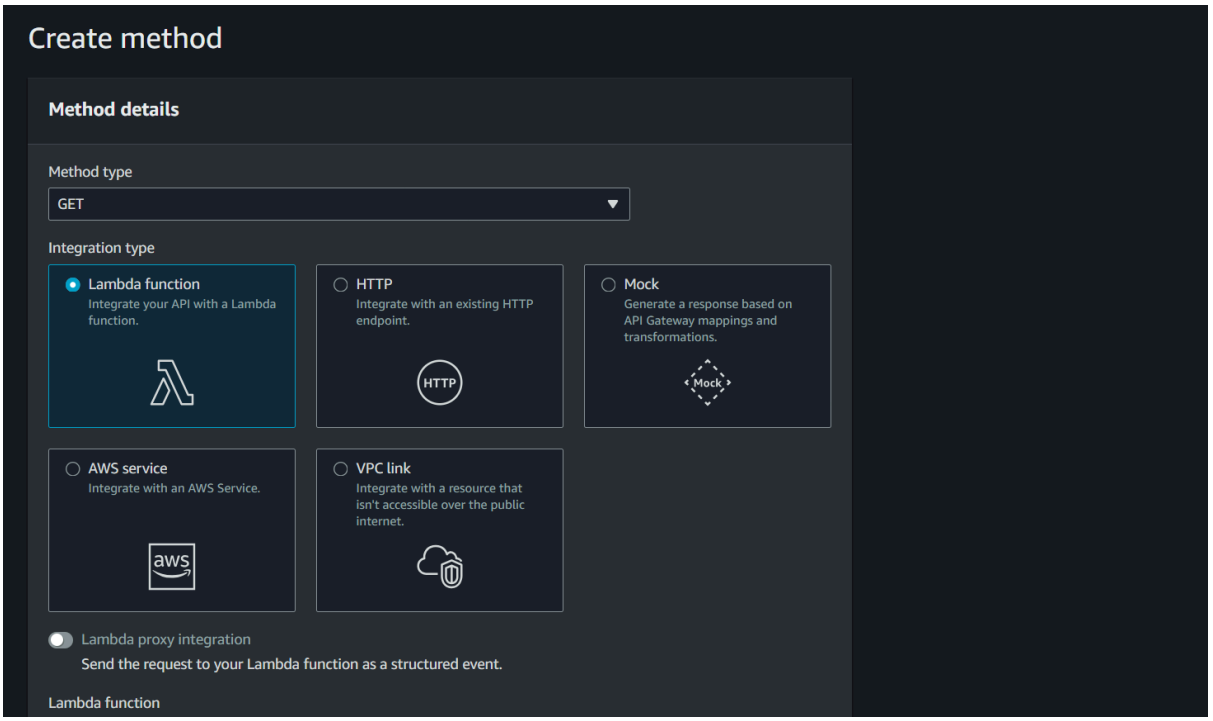
Working fine…

Get Method

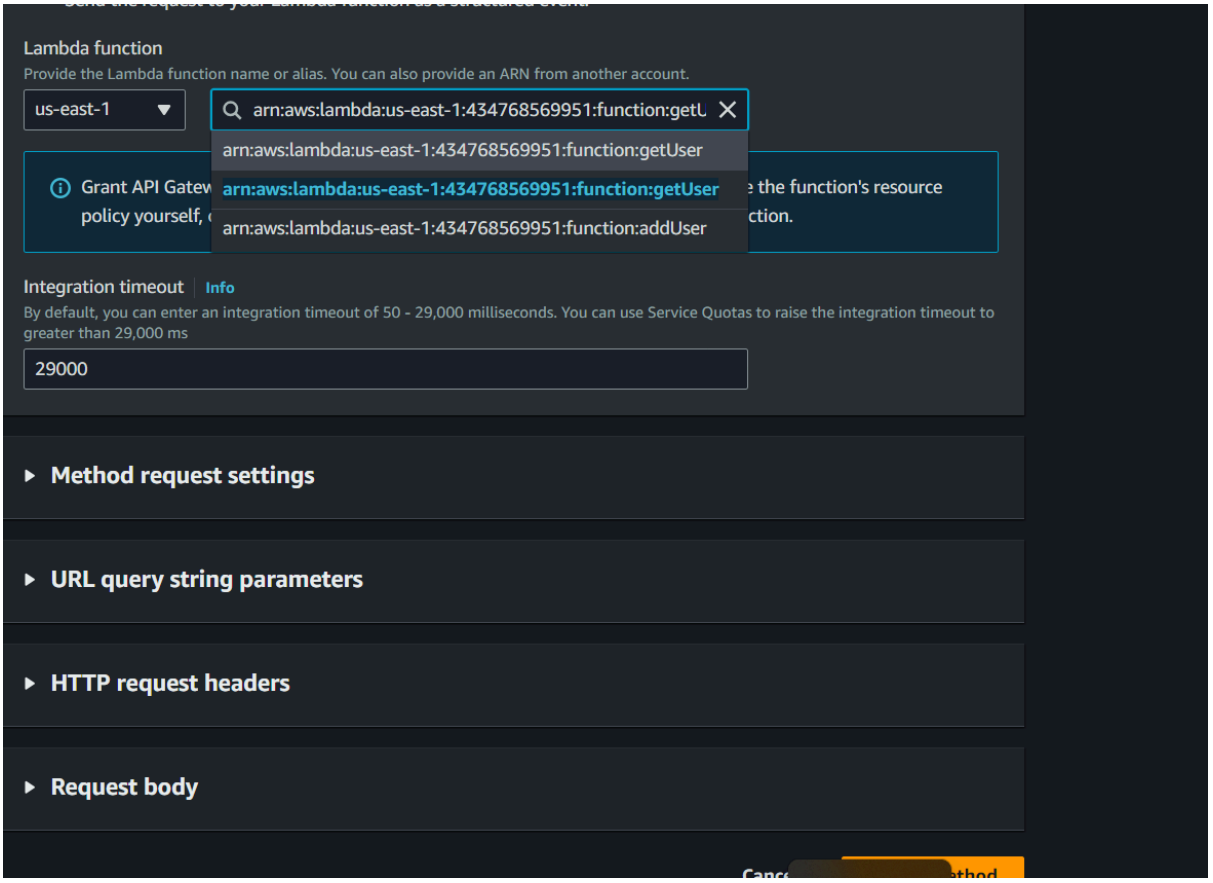
Now click on the / in the left pane again and again click on the create Method (in the right hand side)



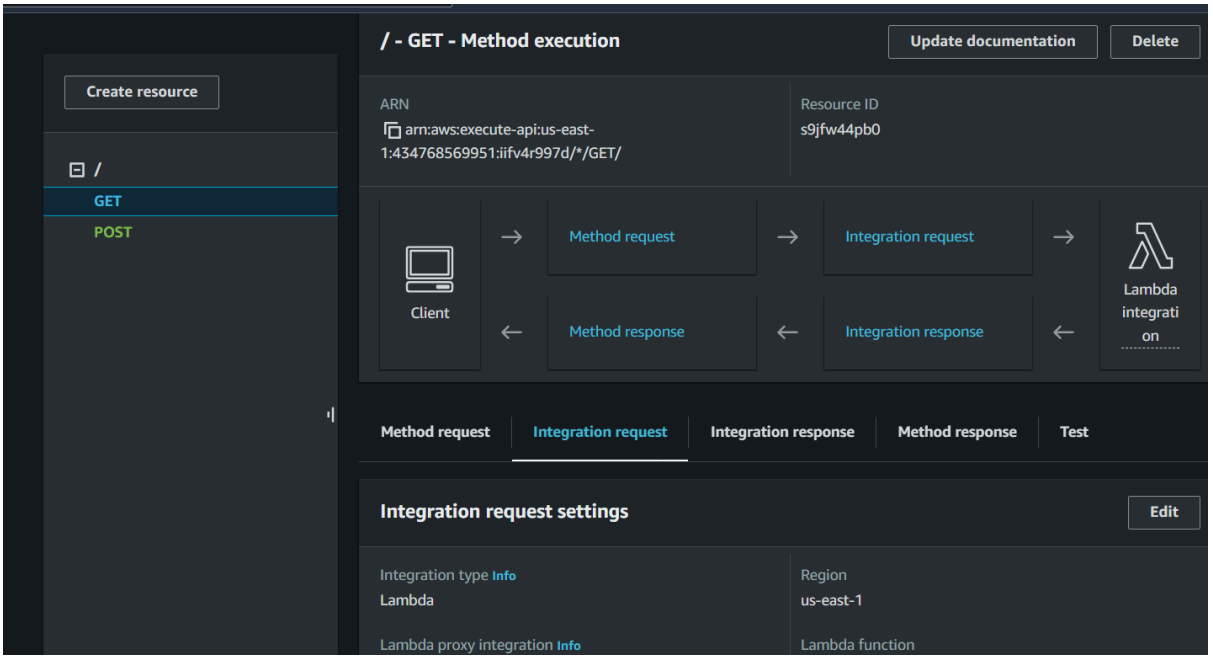
Now select GET as the method type, choose lambda.



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

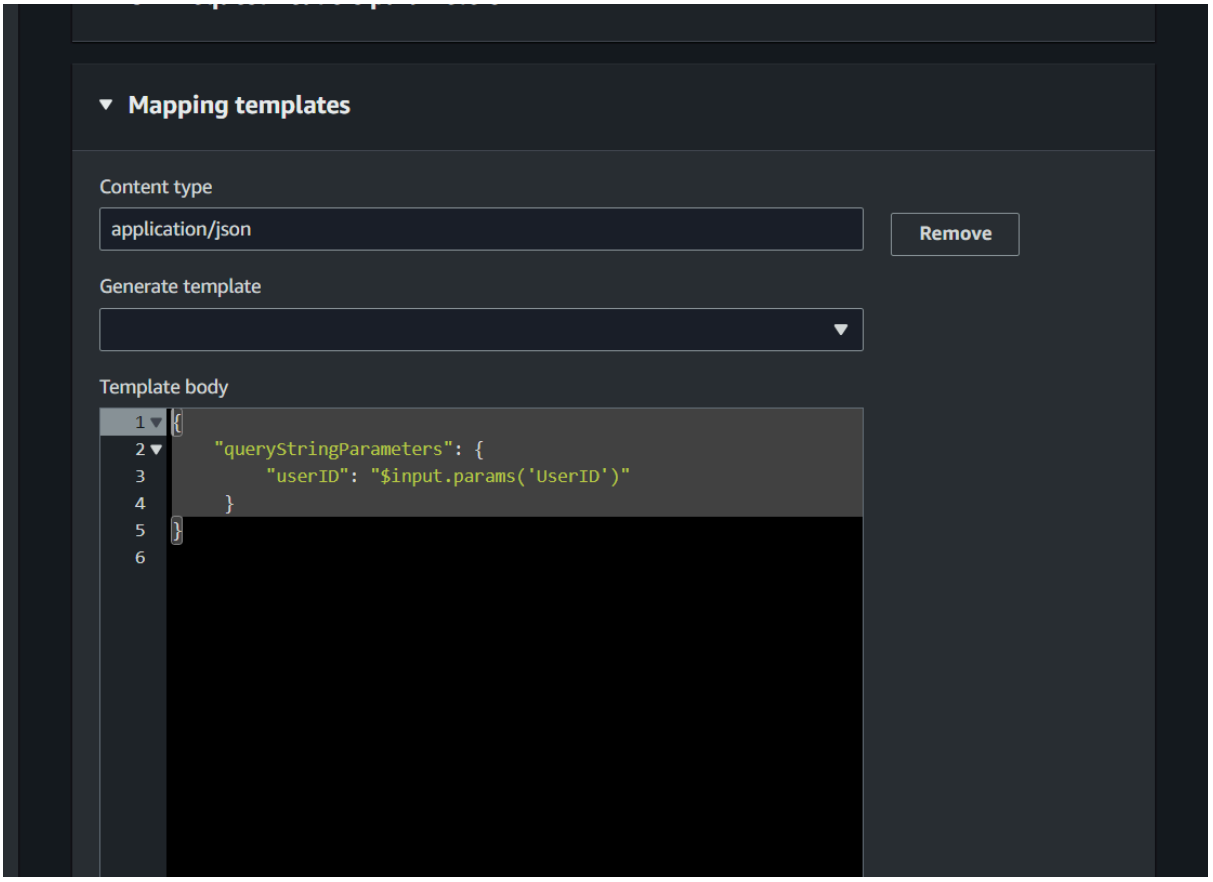


Now Before testing one important step is there:
Go to the integration request section…



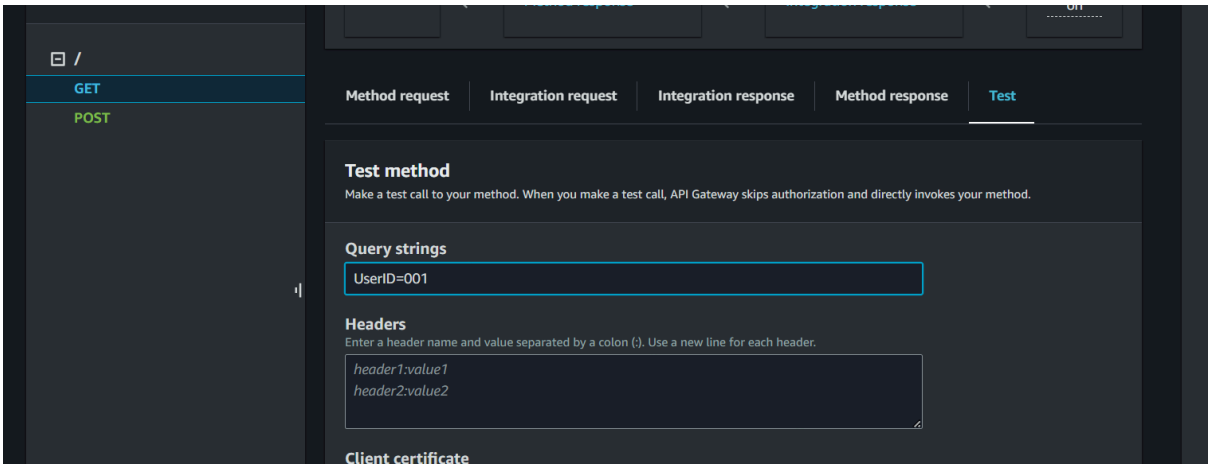
Click on edit.
Scroll down> click on mapping templates> then click on add mapping template
Write “application/json” in content type.
And write this code in the Template body:

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



Click on save.

Now navigate to the test section of this GetMethod and paste the query string here:
Eg:
My query string :
UserID=001



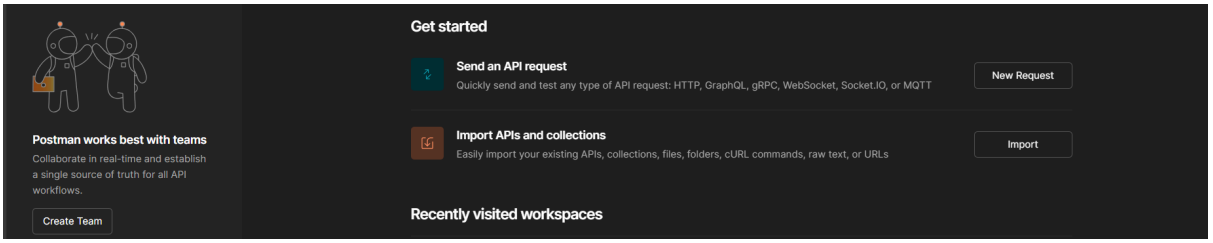
Results:

works fine…

Click on deploy..

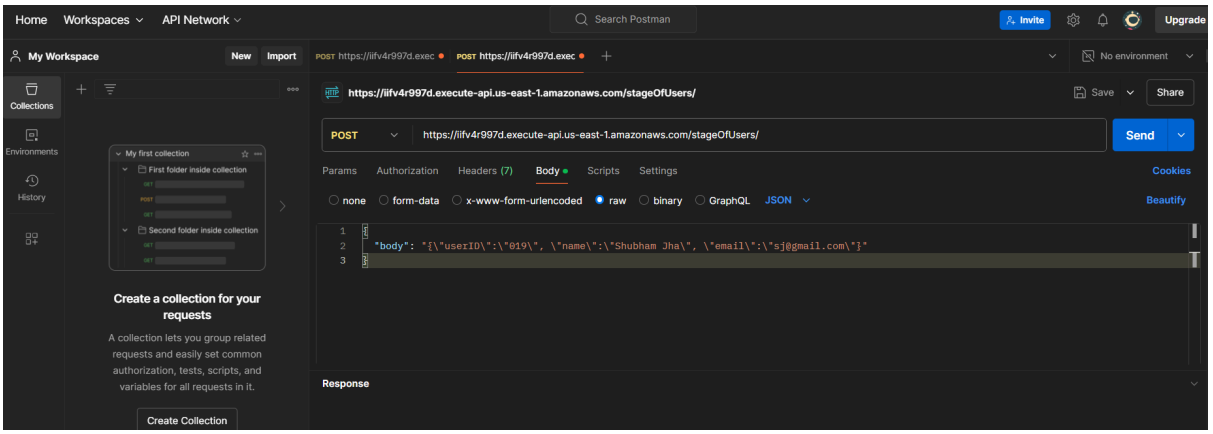
Testing the API (Using postman web)

Open postman web sign in with your account.
Click on send an API request

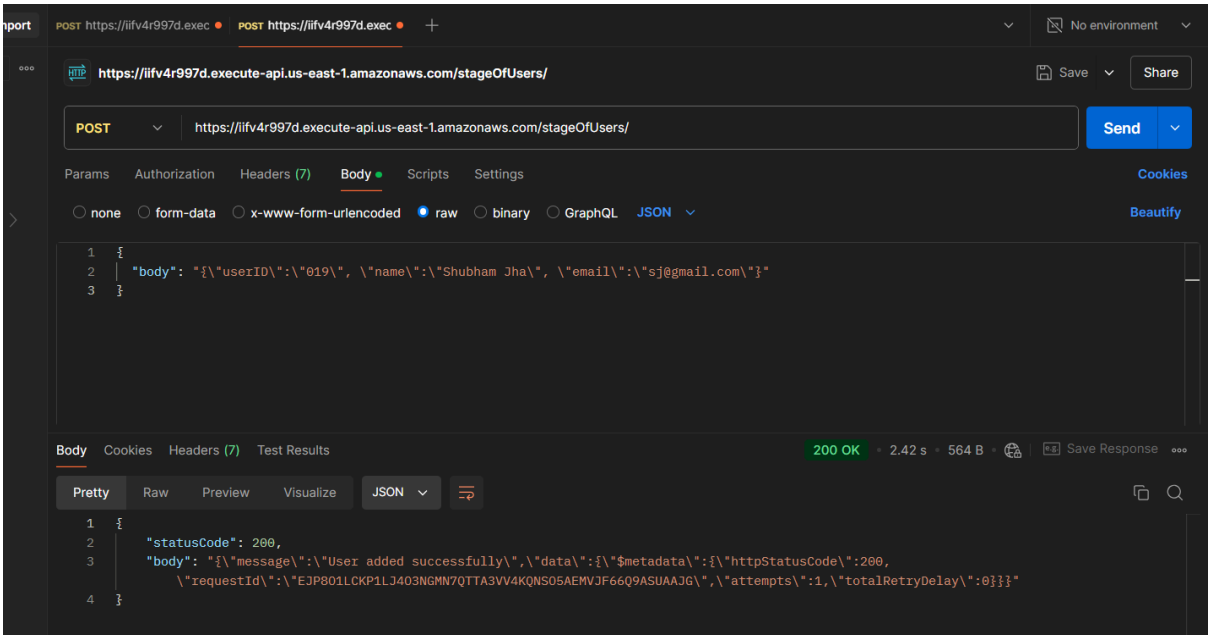


Paste the API link that you got after clicking deploy.
Then click on body, select raw and paste the body part of the test that you wrote (event Json) in the addUser Lambda function.

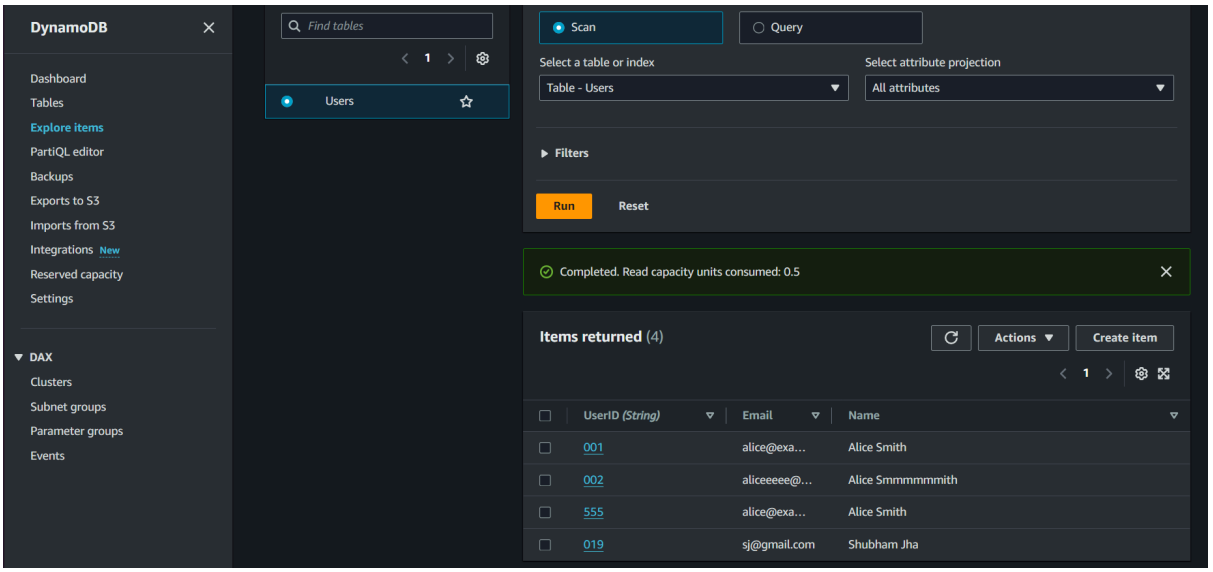
Eg:



Click on send,



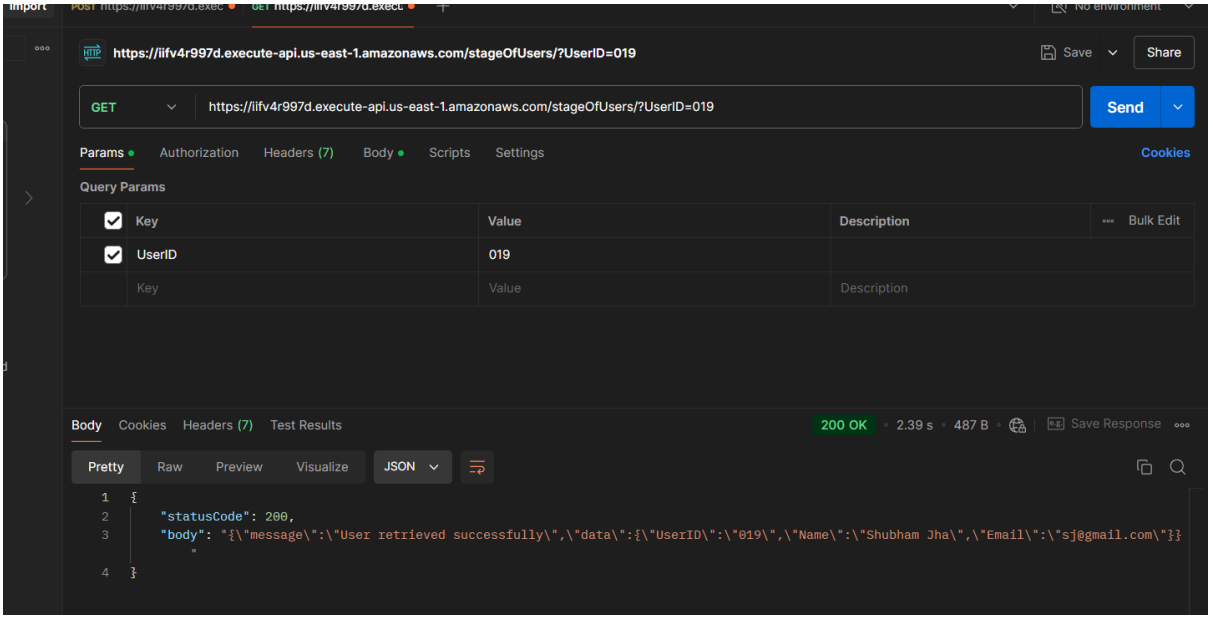
Check the table:



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.