

ADVANCE DEVOPS CASE STUDY REPORT

CASE STUDY TOPIC:

Case Study No:19:Building a Serverless REST API

- **Concepts Used:** AWS Lambda, API Gateway, DynamoDB
- **Problem Statement:** "Create a simple serverless REST API using AWS Lambda and API Gateway to manage user data in a DynamoDB table. The API should support adding a new user and retrieving user details."
- **Tasks:**
 - Create a DynamoDB table to store user data.
 - Write two Lambda functions: one for adding a user to the table and another for retrieving user details by ID.
 - Set up an API Gateway to trigger these Lambda functions based on HTTP methods (POST for adding and GET for retrieving).
 - Test the API using curl or Postman.

INTRODUCTION:

Case Study Overview:

- This case study focuses on building a serverless REST API using AWS Lambda, API Gate way, and DynamoDB. The goal is to manage user data effectively by creating a simple, scalable, and efficient solution without the need for traditional server management.

Key Feature and Application:

- **Unique Feature:** The serverless architecture eliminates the need for server provisioning and management. It scales automatically based on demand, reducing costs and operational complexity.
- **Practical Use:** This setup is ideal for applications requiring scalable APIs with minimal infrastructure management, such as mobile backends, microservices, and realtime data processing.

1.Step-by-Step Explanation:

Step 1: Initial Setup - Creating a DynamoDB Table

- **Log in to AWS Management Console.**
- **Open DynamoDB from the services.**
- **Create a Table:**
 - Table Name: Users
 - Partition Key: UserId (String)
 - Leave other settings as default.
 - Click Create Table

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.

Users

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.

Userid

String

1 to 255 characters and case sensitive.

Sort key - optional

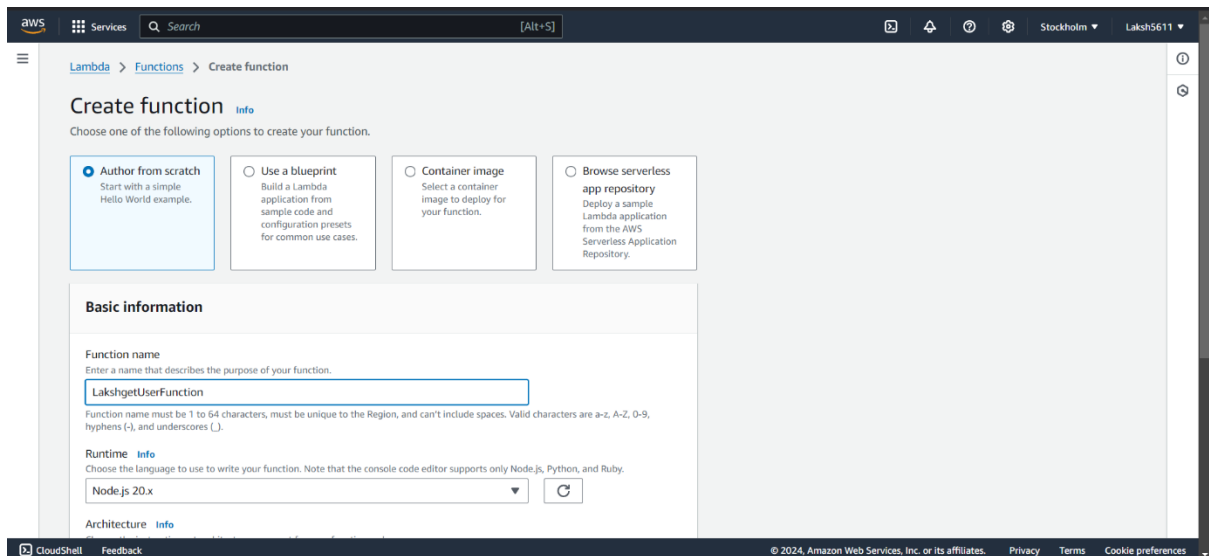
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name

String

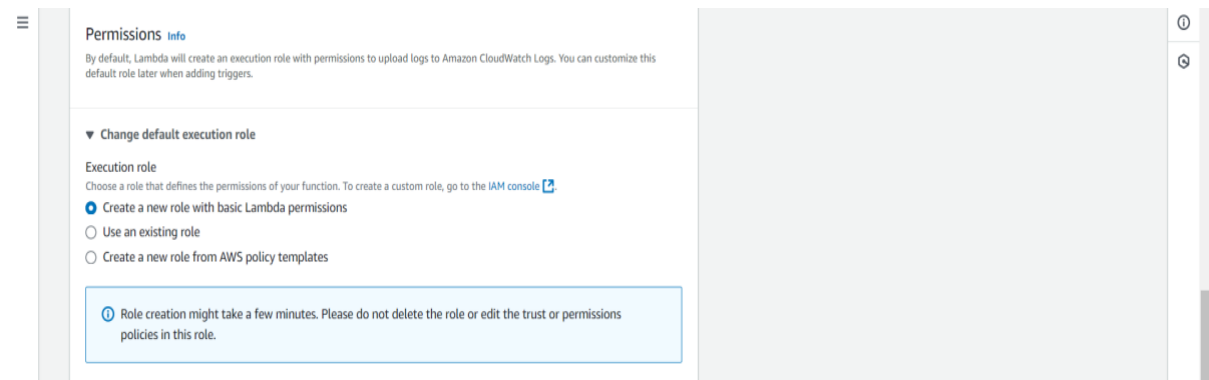
1 to 255 characters and case sensitive.

1. **Open AWS Lambda from the services.**
2. **Click Create Function.**
3. **Choose Author from scratch**
 - Function Name: addUserFunction
 - Runtime: Python 3.12

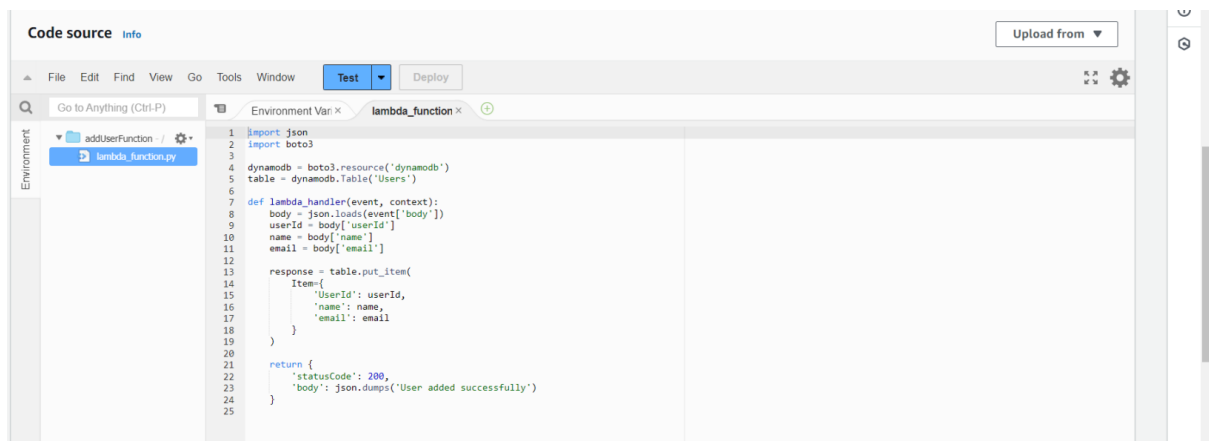


4. Create new role with permission to access DynamoDB.

5. Click Create Function

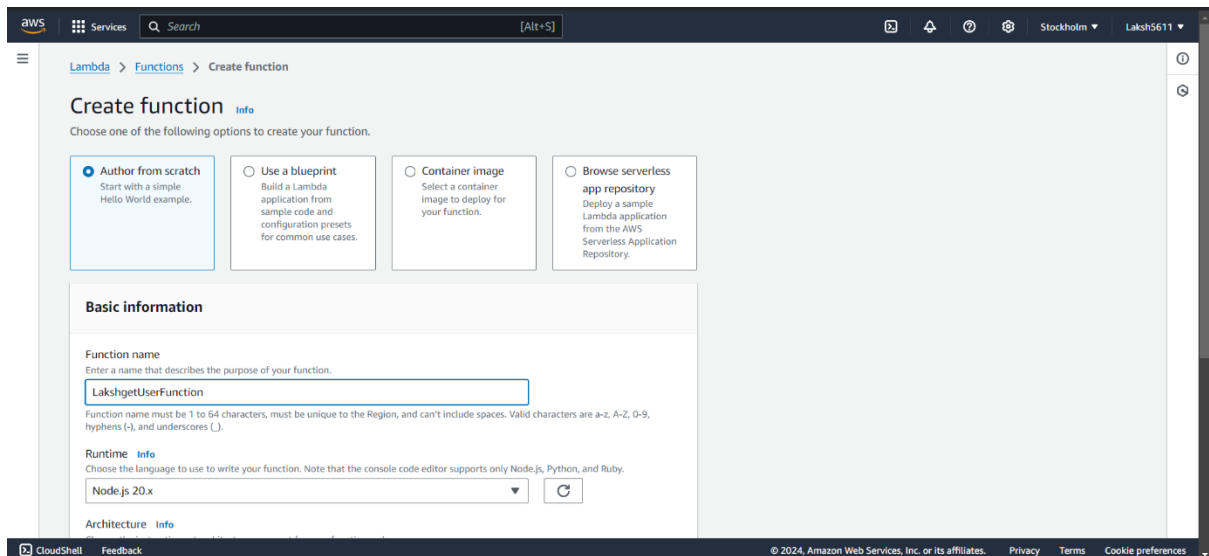


Code for addUserFunction:

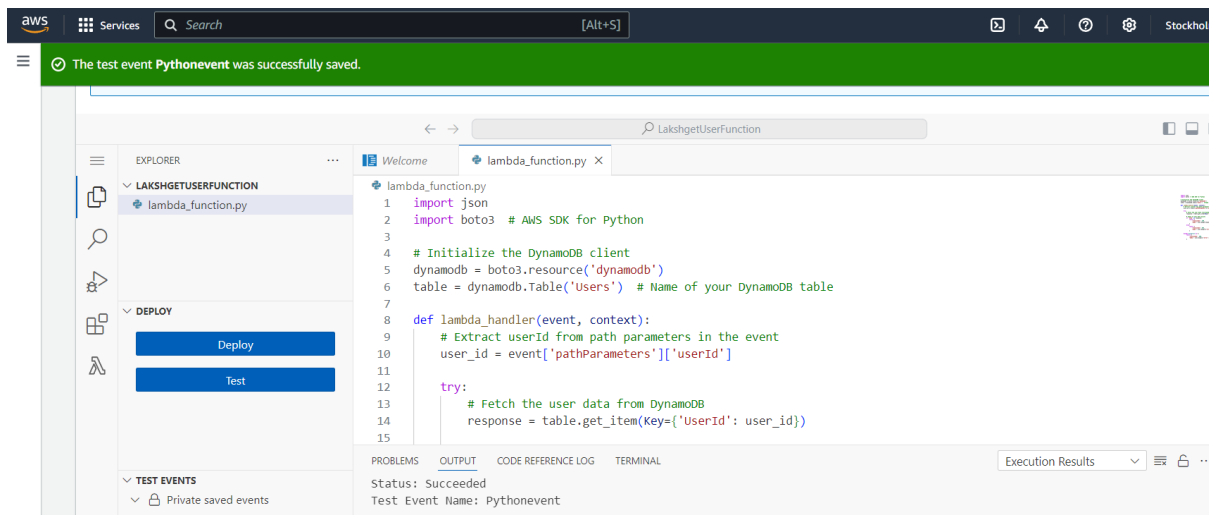


Creating getUserFunction

1. Repeat the process to create another Lambda function.
 - Function Name: getUserFunction
 - Runtime: Python 3.X
 - Select the same execution role that has DynamoDB access



Code for getUserFunction:

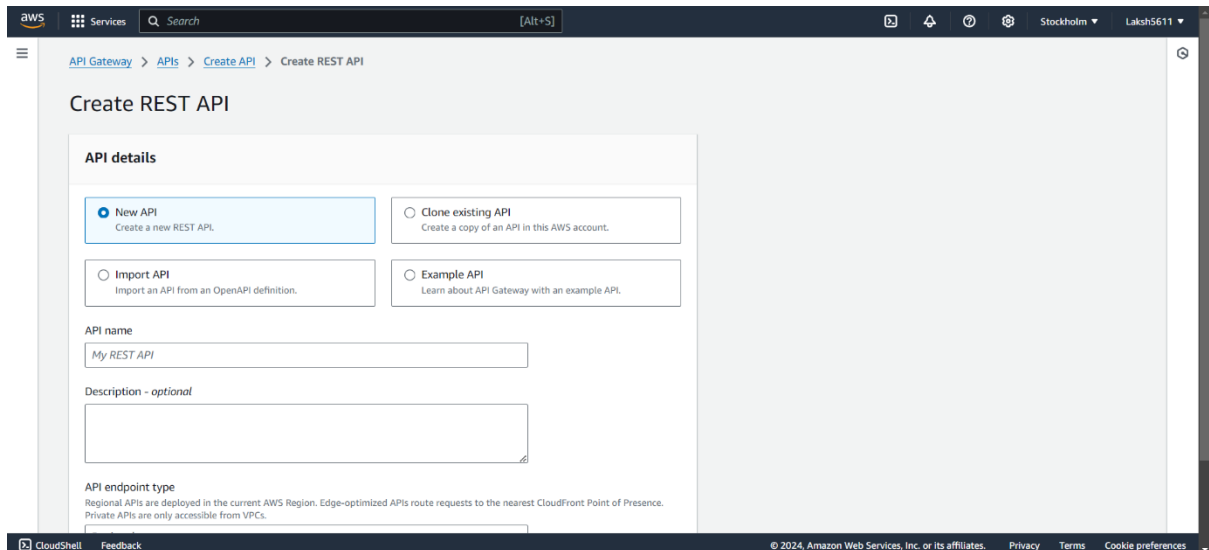


Output-



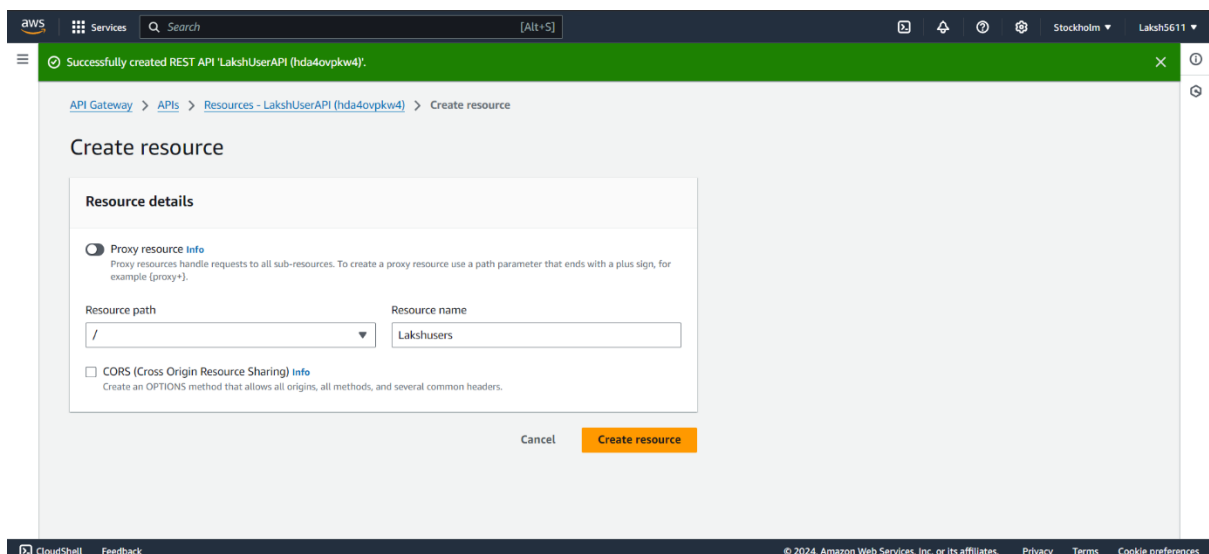
Step 3: API Gateway Configuration Creating the API

1. Open API Gateway from the services.
2. Click Create API.
3. Choose HTTP API or REST API.
4. Name it (e.g., UserAPI).

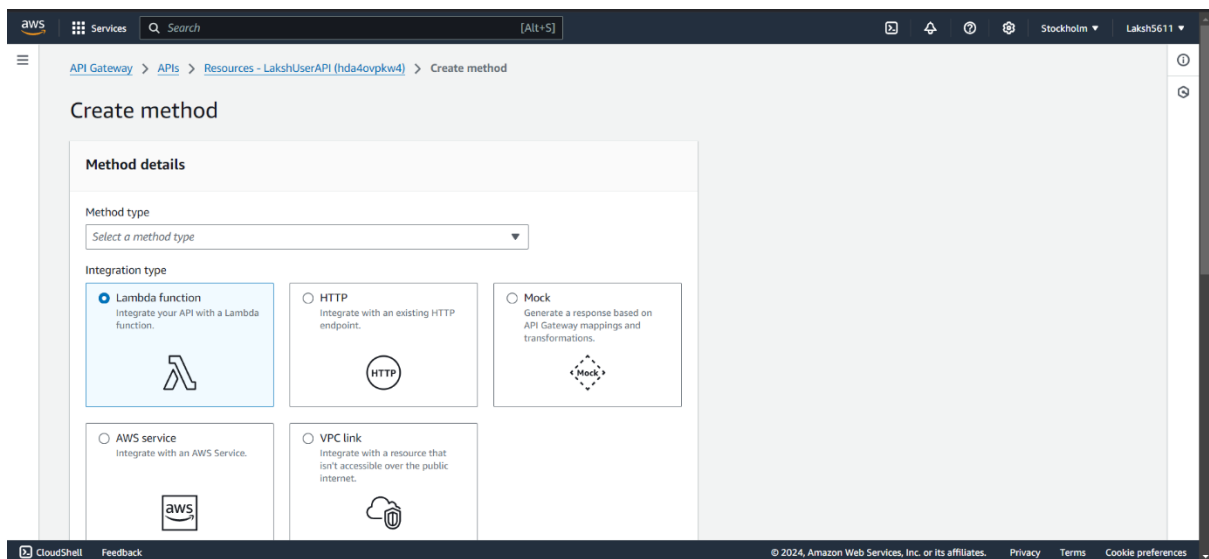


Creating POST Method for Adding User

1. Inside API Gateway, click Actions > Create Resource.
2. Provide a Resource Name (e.g., user).
3. Keep the Resource Path as /user.
4. Click Actions > Create Method.
5. Select POST.

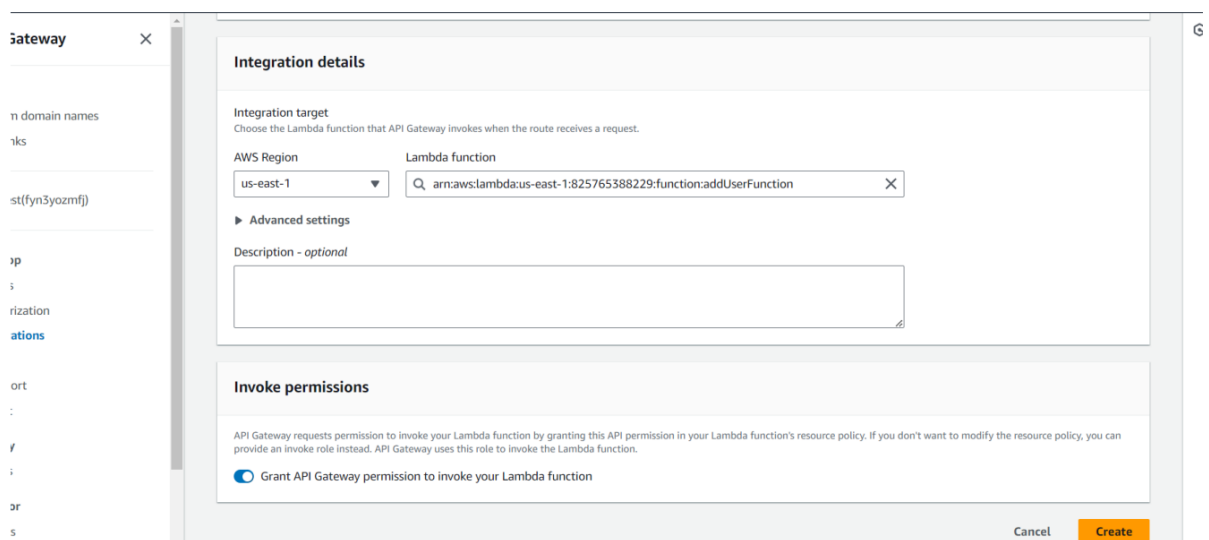


6. Under Integration Type, choose Lambda Function



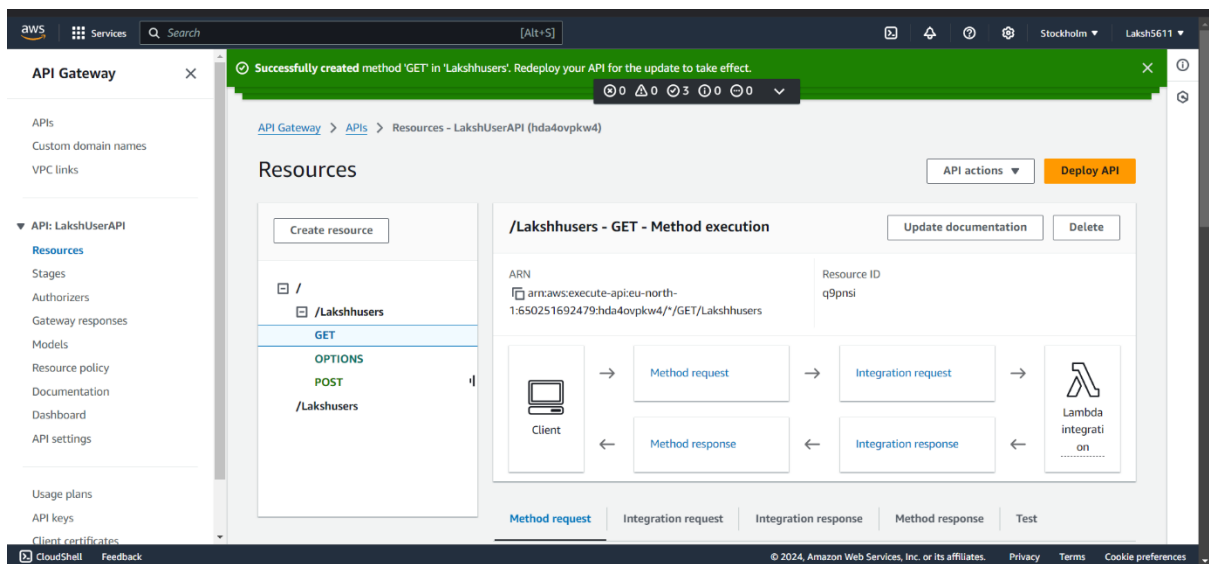
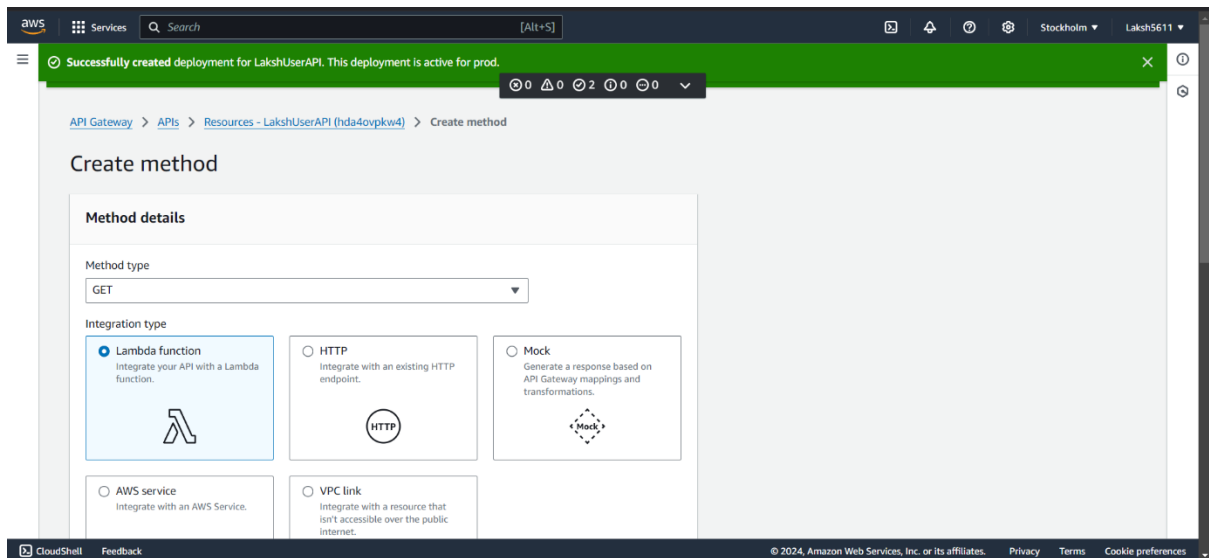
7. Select the addUserFunction Lambda function.

8. Click create.

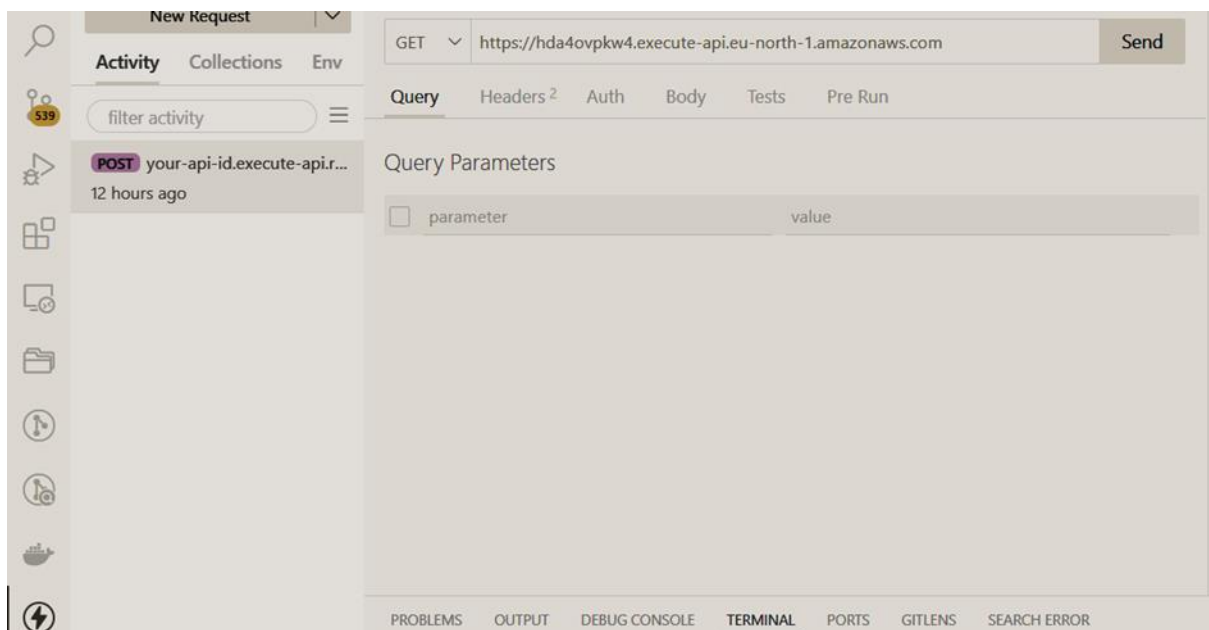


Creating GET Method for Retrieving User by ID

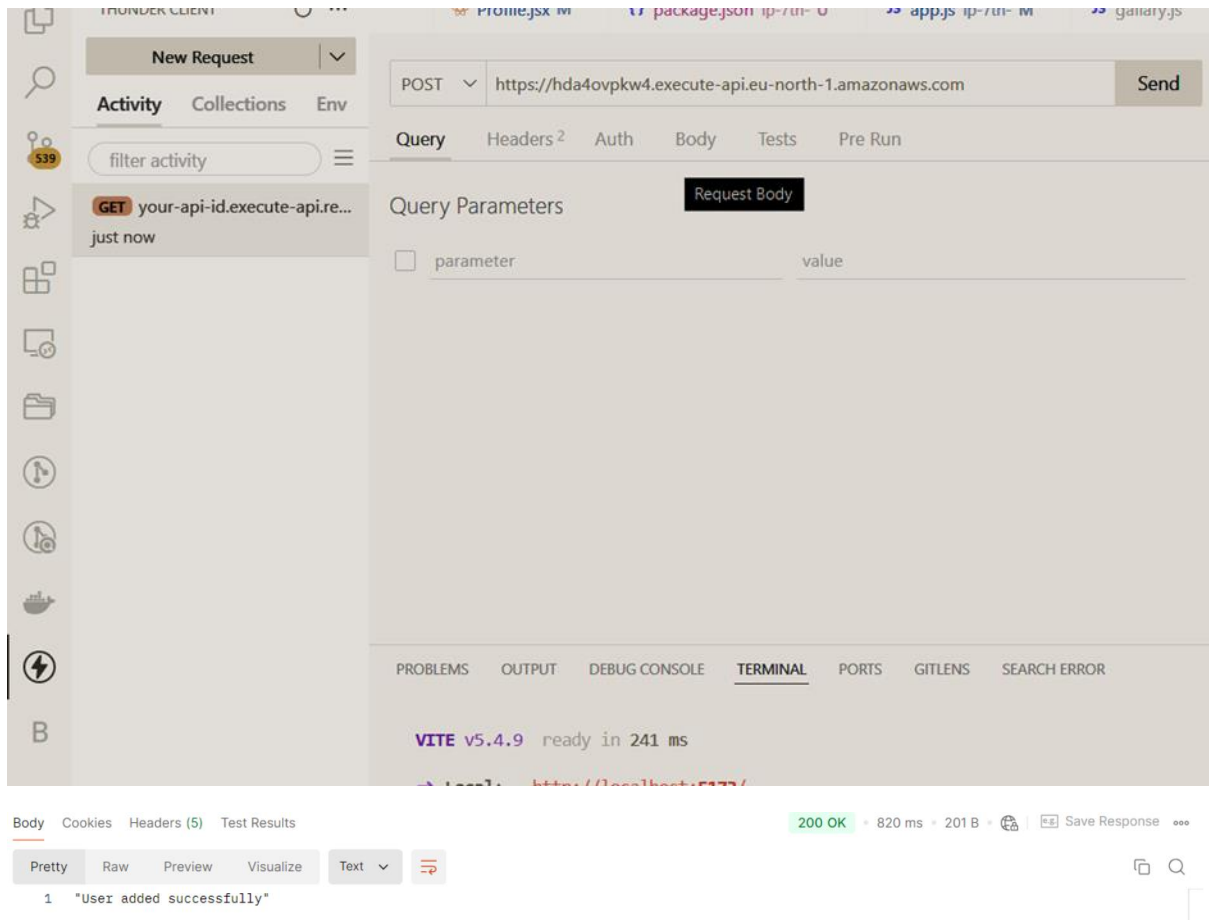
1. Under Actions, select Create Method.
2. Choose GET.
3. Under Integration Type, choose Lambda Function.
4. Select the getUserFunction Lambda function.
5. Set Path Parameters (e.g., /user/{userId}).
6. Click Create.



Testing the api on Thunder Client(V.S. Code):

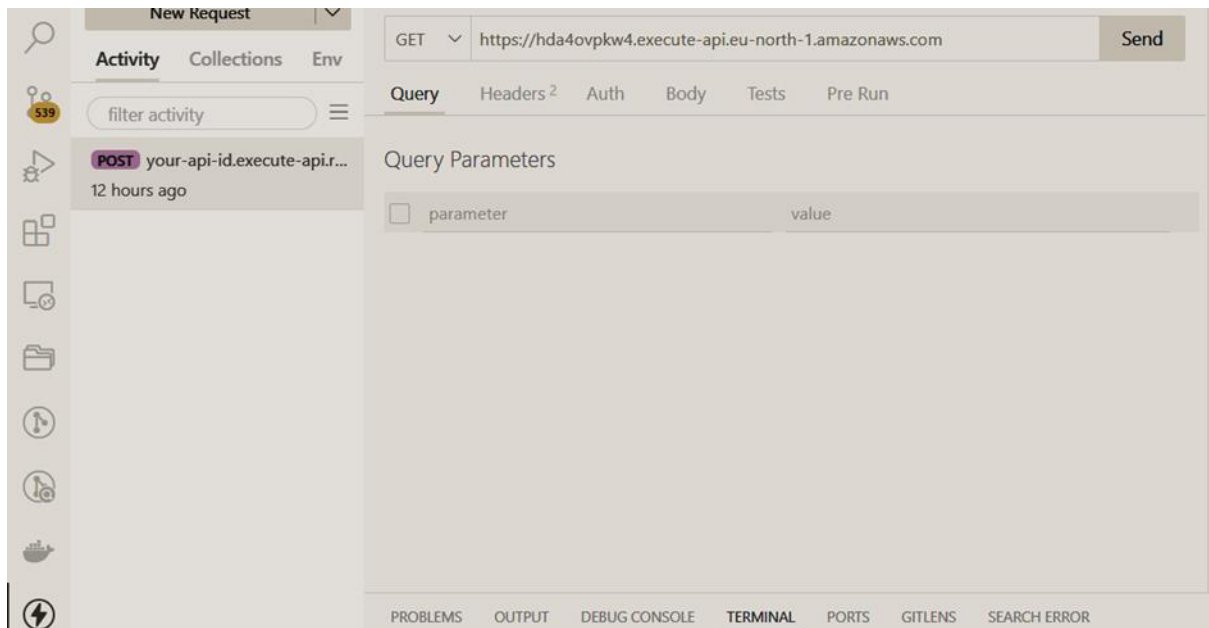


Post request to add data to the dynamo table



The screenshot shows the Thunder Client interface. On the left, the 'Activity' tab is selected, showing a list of requests. The first request is a GET request to 'your-api-id.execute-api.eu-north-1.amazonaws.com' with a status of 'just now'. The main panel displays a POST request to 'https://hda4ovpkw4.execute-api.eu-north-1.amazonaws.com'. The 'Query' tab is selected, showing a table for 'Query Parameters' with columns 'parameter' and 'value'. The 'Request Body' tab is also visible. At the bottom, the 'Terminal' tab shows the output of a Vite command: 'VITE v5.4.9 ready in 241 ms'. The status bar at the bottom indicates a '200 OK' response with a status of '200 OK', a time of '820 ms', and a size of '201 B'.

Get request to fetch the user data from table



The screenshot shows the Thunder Client interface. On the left, the 'Activity' tab is selected, showing a list of requests. The first request is a POST request to 'your-api-id.execute-api.eu-north-1.amazonaws.com' with a status of '12 hours ago'. The main panel displays a GET request to 'https://hda4ovpkw4.execute-api.eu-north-1.amazonaws.com'. The 'Query' tab is selected, showing a table for 'Query Parameters' with columns 'parameter' and 'value'. The 'Request Body' tab is also visible. At the bottom, the 'Terminal' tab shows the output of a Vite command: 'VITE v5.4.9 ready in 241 ms'. The status bar at the bottom indicates a '200 OK' response with a status of '200 OK', a time of '820 ms', and a size of '201 B'.

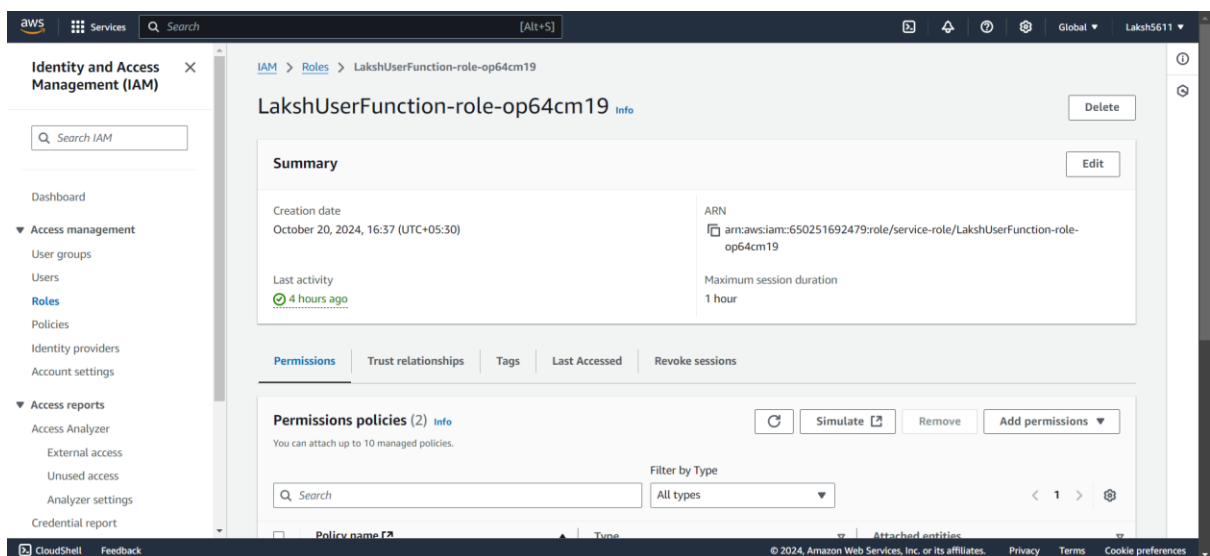
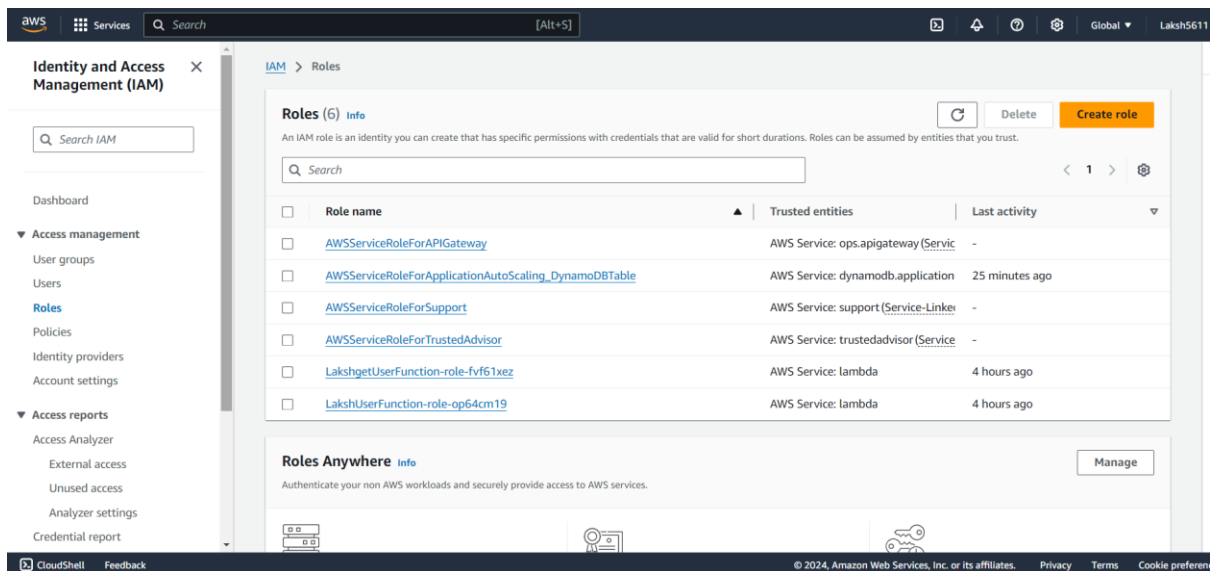
Trying to fetch the data not present in the table

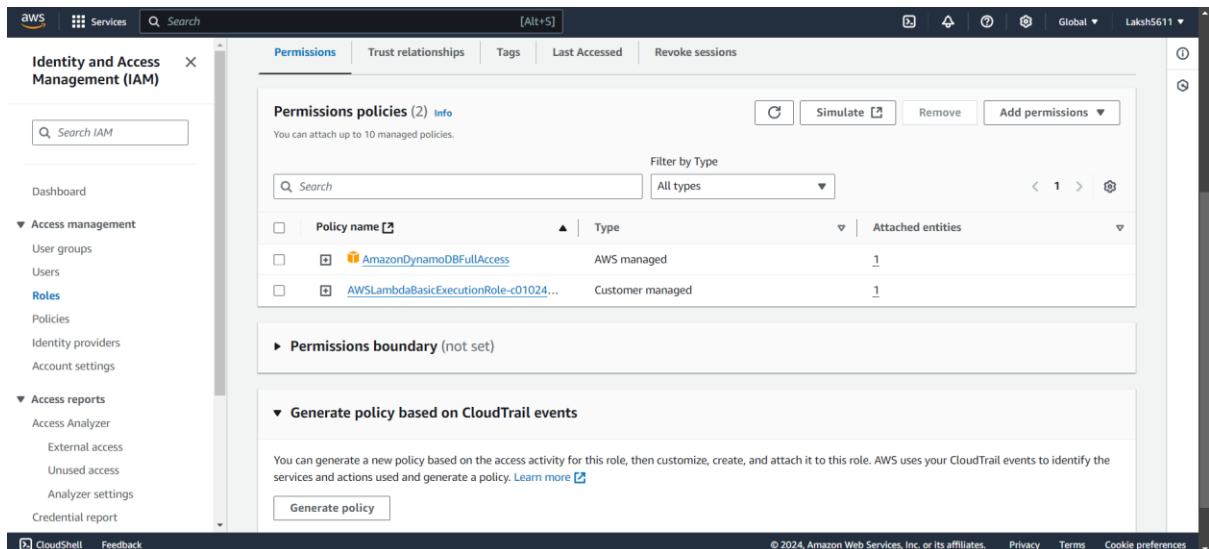


Guidelines

- Ensure IAM roles have correct permissions:

- Attach DynamoDBAccessPolicy with actions: dynamodb:PutItem and dynamodb:GetItem
- Attach AWSLambdaBasicExecutionRole for logging.





Problems I Faced During execution :

Problem 1: Permissions Error

- **Issue:** Encountered Access Denied Exception when the Lambda function tried to perform operations on DynamoDB.
- **Solution:** Updated the IAM role to include the necessary permissions.

Attached the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:GetItem"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:825765388229:table/Users"
    }
  ]
}
```

Problem-2: Lambda Function Errors (Node.js/ES Modules)-

- **Problem:** Initially, you encountered errors related to using `require` in a Node.js Lambda function. The error message was: `require is not defined in ES module scope, you can use import instead`.
- **Solution:** To resolve this, you had to change your Lambda function syntax to use ES module imports (`import`) instead of the CommonJS `require`. This was necessary because AWS Lambda was using an environment that defaults to ES modules.

Problem-3: Handling POST and GET Method Differences-

- **Problem:** There was confusion about setting up events for both GET and POST methods in the Lambda function. You needed clarity on how to differentiate between the payloads for the two types of requests.
- **Solution:** For GET requests, the event payload came from query parameters, while for POST, the data came from the request body. You needed to create different test event payloads accordingly, with the POST event requiring a JSON body that reflects user input like Name and Email.

Conclusion:

In this case study, I have successfully built a Serverless REST API using AWS Lambda, API Gate way, and DynamoDB. The key features include the serverless architecture that eliminates the need for server management, and the scalability and cost-efficiency provided by AWS services.

Throughout the project, we tackled several challenges, including configuring permissions, ensuring correct setup of the Lambda functions, and managing costs. By leveraging these technologies, we demonstrated how to efficiently manage user data with minimal overhead and high reliability.

This project showcases the practical application of cloud-

native solutions in building robust and scalable APIs, emphasizing the importance of proper configuration and diligent cost management.

The implementation of this Serverless REST API is a testament to the power of cloud services in creating efficient, scalable, and cost-effective solutions for modern applications