SNEHAL A. PATIL D15A 39

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, scal able, 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 an d management. It scales automatically based on demand, reducing costs and operational c omplexity.
- **Practical Use**: This setup is ideal for applications requiring scalable APIs with minimal infra structure management, such as mobile backends, microservices, and real-time data processing.

Third-Year Project Integration :

Integrating Serverless REST API in Glamease Salon Appointment Management System

In Glamease, we can leverage the serverless architecture (AWS Lambda, API Gateway, and DynamoDB) to streamline various functionalities. This integration will make the system more scalable, cost-effective, and easier to maintain without managing traditional servers.

1. User and Salon Registration

- **How**: The **Lambda function** can handle both user and salon registration. When a user or salon admin registers, the registration form data is sent via a **POST request** through **API Gateway** to the Lambda function.
- **Integration**: This allows storing the user/salon details in a **DynamoDB table**. The system can then fetch and display this data in the admin dashboard using a **GET request** to the same API.

2. Salon Admin Dashboard: Viewing Profiles

- **How**: Admins can view profiles using another Lambda function integrated with **GET requests**. This API retrieves user or salon information from **DynamoDB** and displays it on the admin dashboard.
- **Integration**: The admin dashboard triggers the API when accessing user profiles or their own salon details. The serverless nature ensures real-time scalability and reduced maintenance effort.

3. Appointment Booking and Management

- **How**: Users can book appointments through a **POST request** to the API, which sends the appointment data (e.g., date, time, salon ID) to the Lambda function.
- **Integration**: The Lambda function stores appointment details in **DynamoDB**, and salon admins can view/manage appointments by sending **GET requests** to retrieve appointment information.

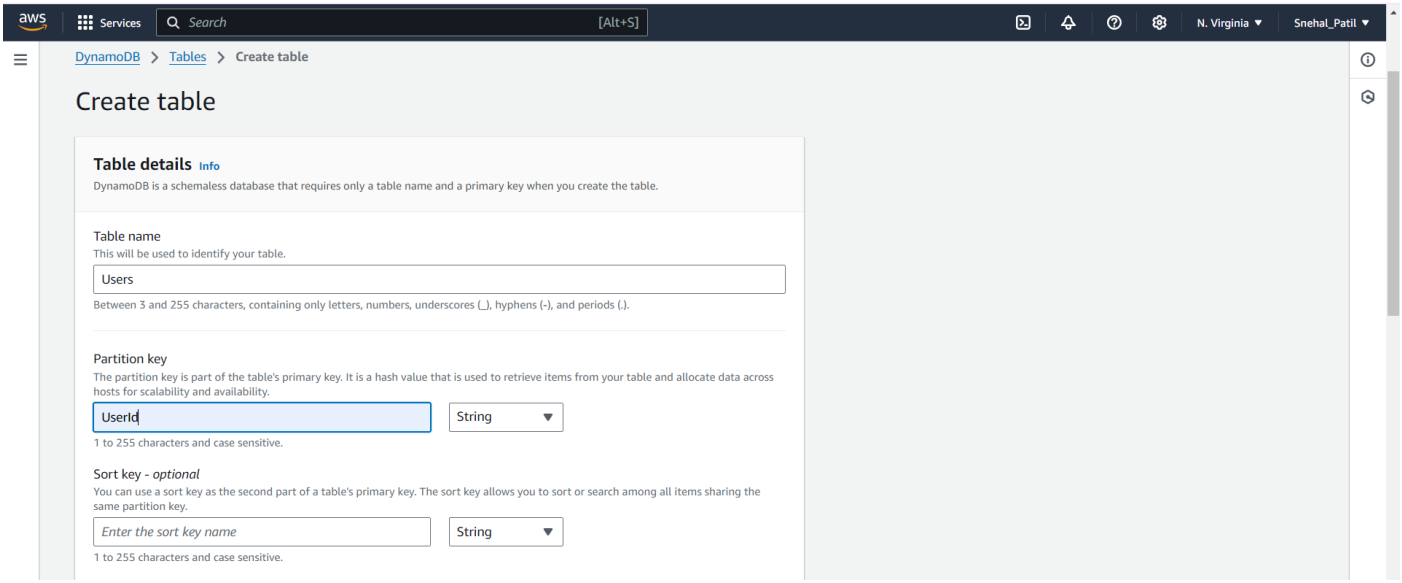
4. Managing Salon Services and Products

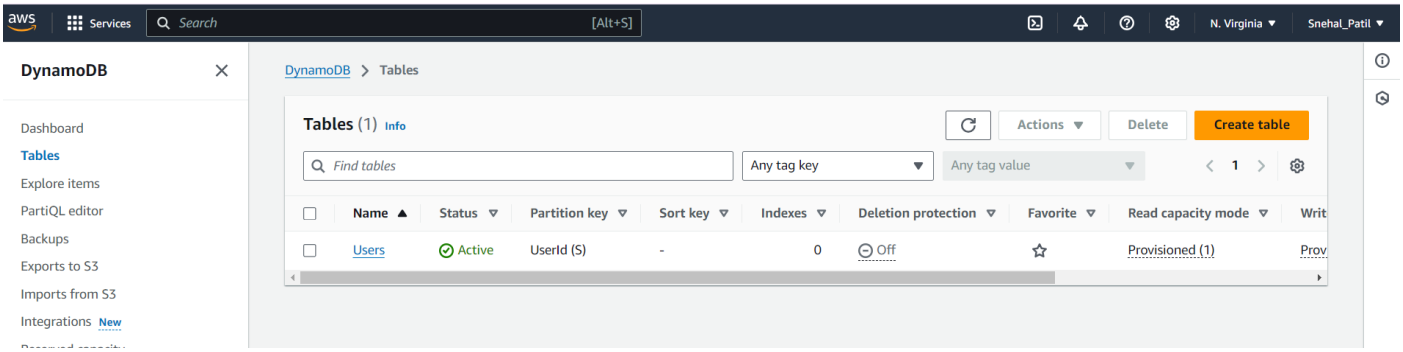
- **How**: Salon admins can use the API to add or update services, products, and packages by sending data through **POST/PUT requests**.
- **Integration**: The Lambda function stores this data in **DynamoDB**, allowing users to view updated services by querying the API.

2. 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**.

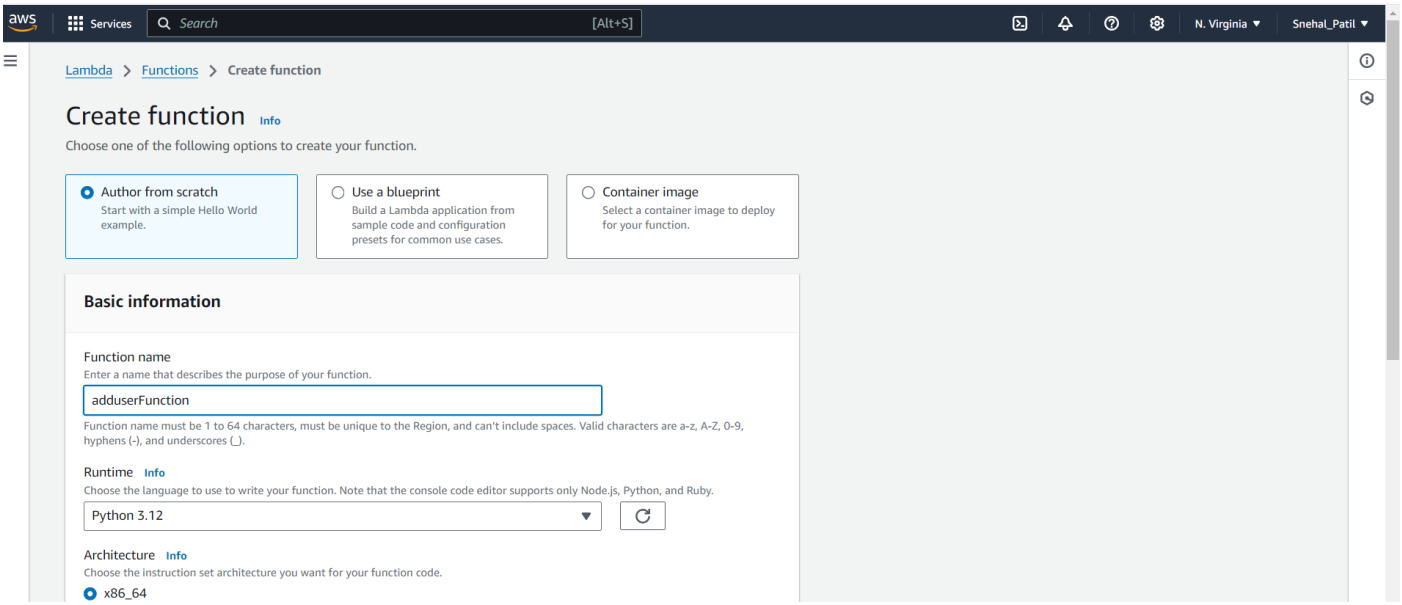




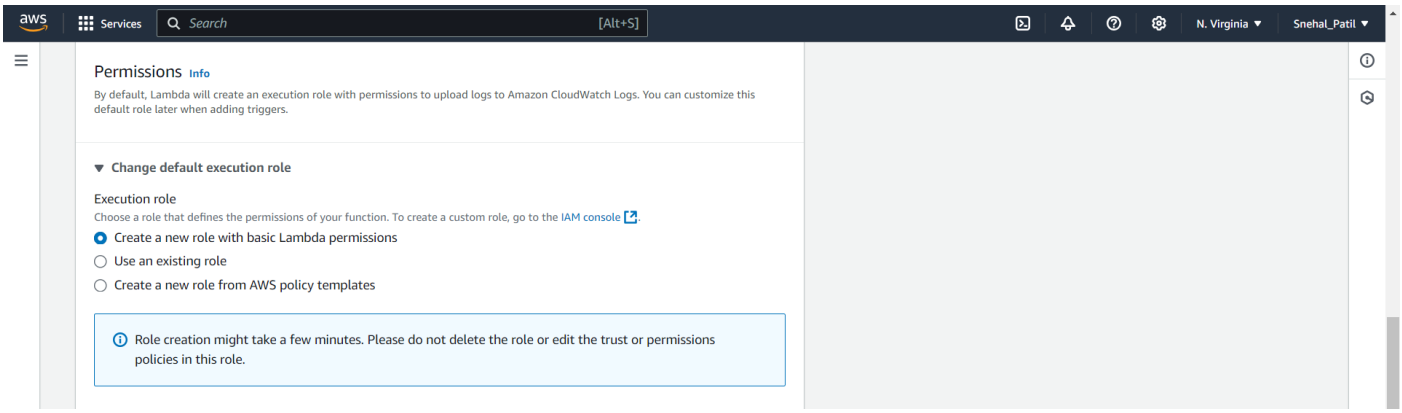
Step 2: Creating AWS Lambda Functions

Creating addUserFunction

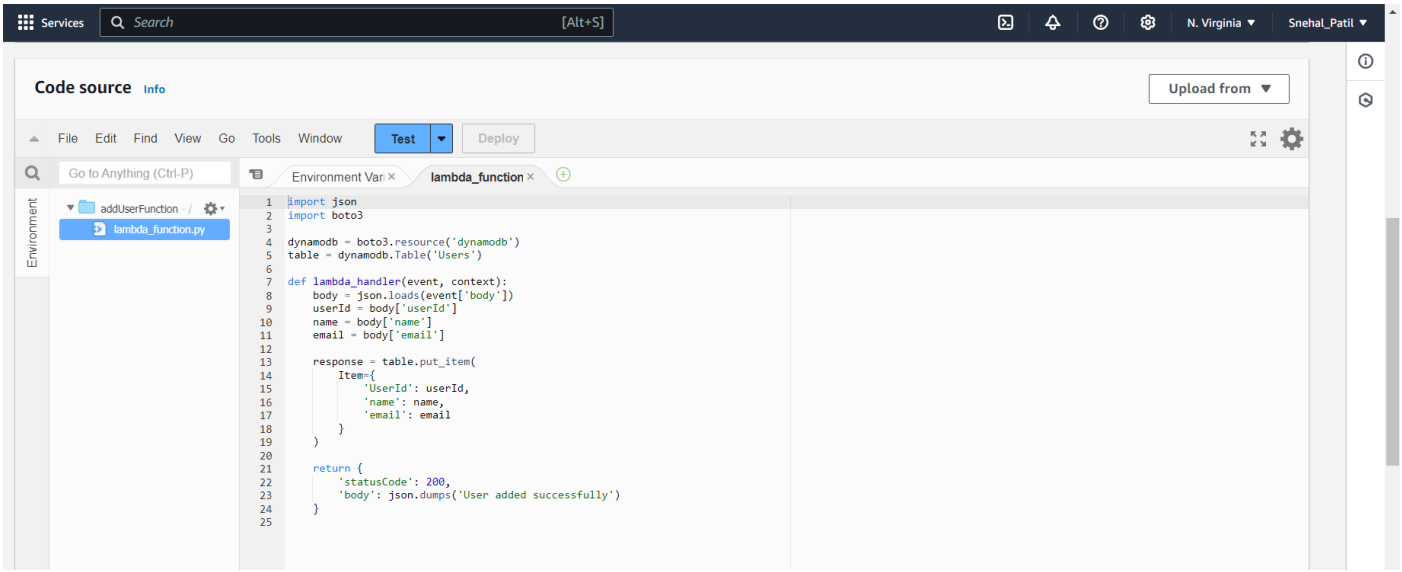
1. Open **AWS Lambda** from the services.
2. Click **Create Function**.
3. Choose **Author from scratch**.
    - Function Name: addUserFunction
    - Runtime: Python 3.12



4. Create new role with permission to access DynamoDB.
5. Click **Create Function**.

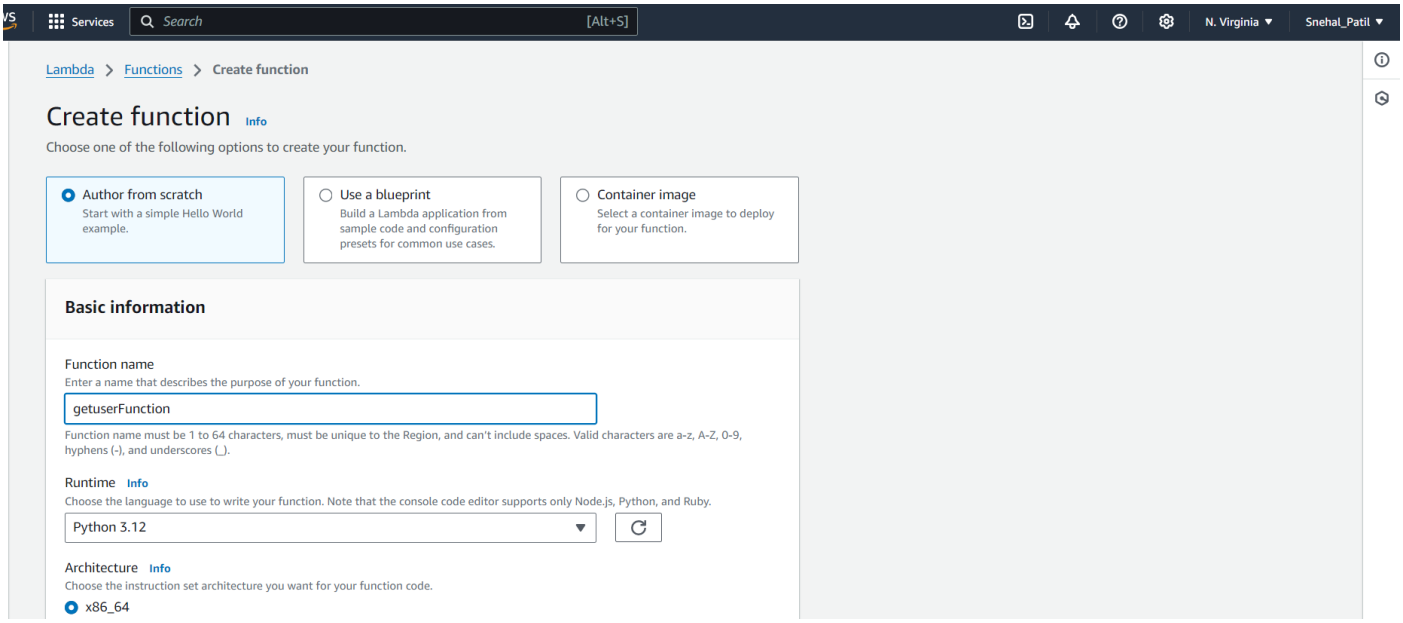


Code for addUserFunction:

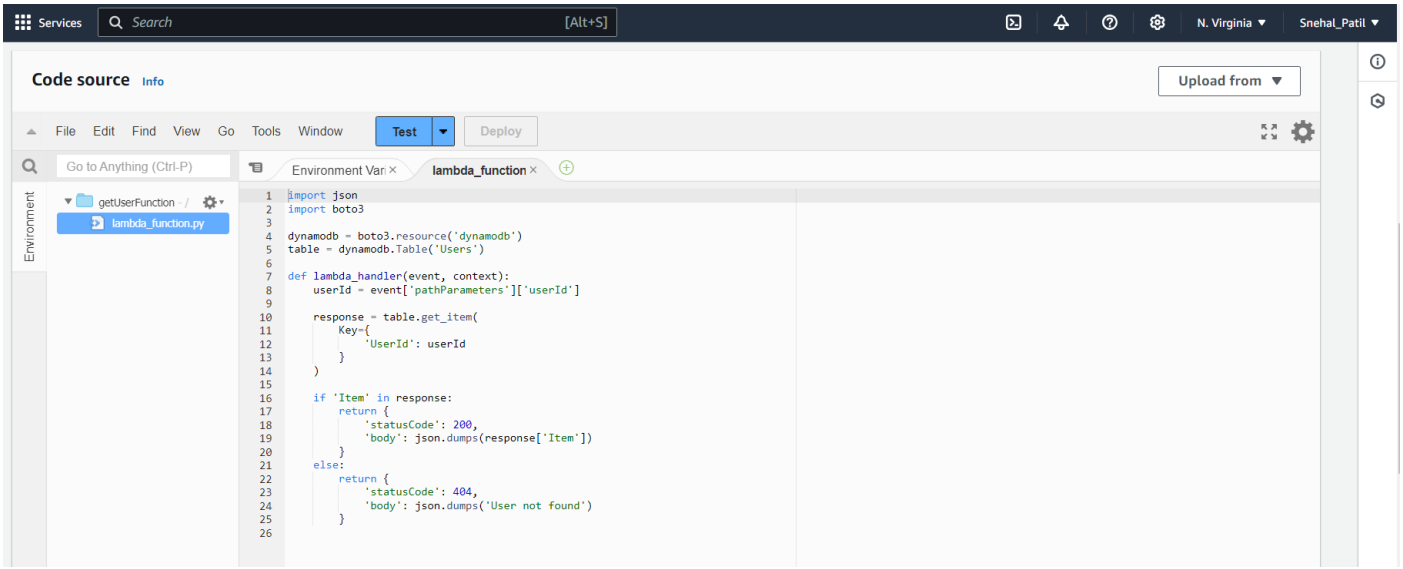


Creating getUserFunction

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.



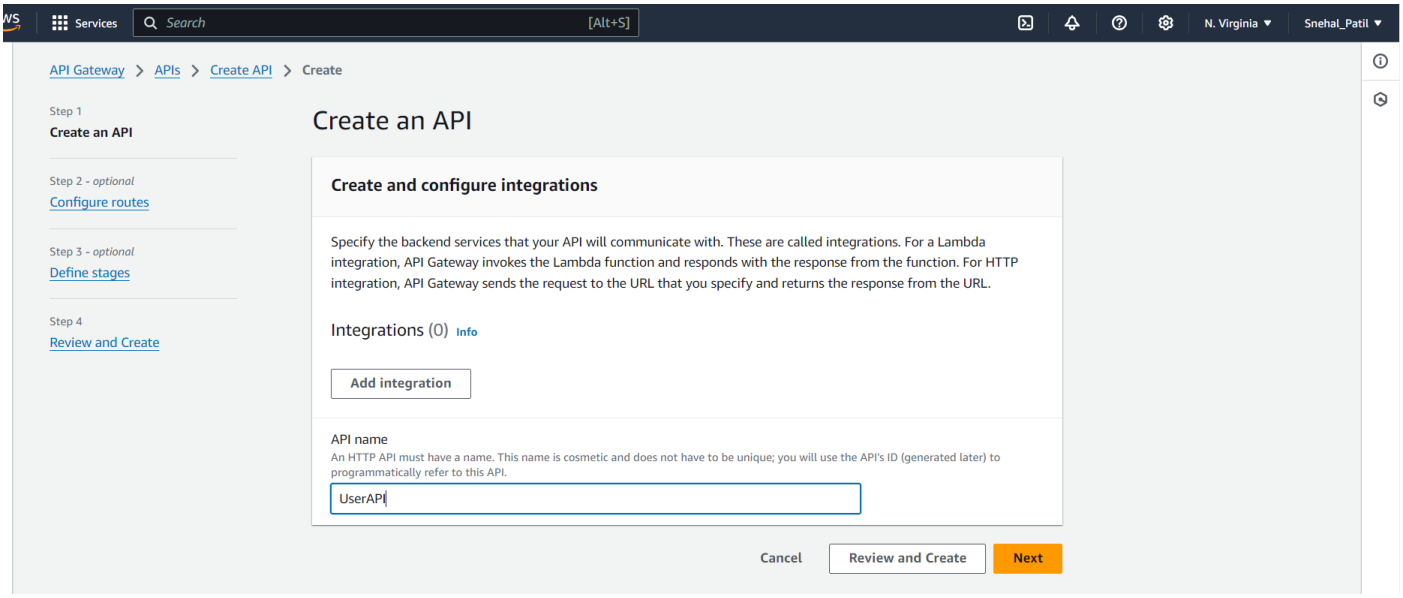
Code for getUserFunction:



Step 3: API Gateway Configuration

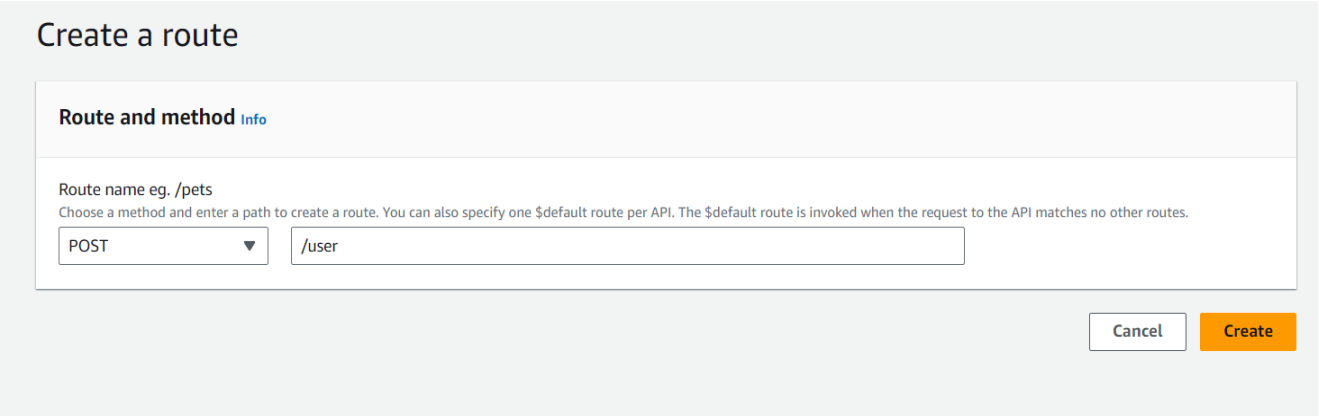
Creating the API

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

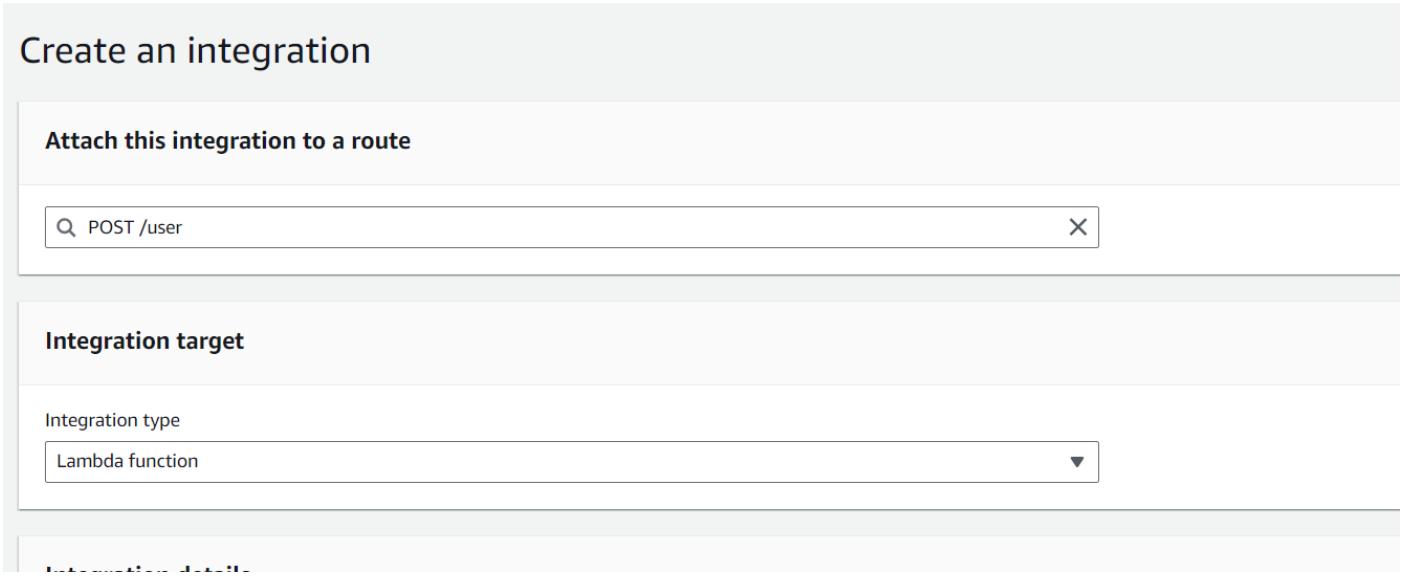


Creating POST Method for Adding User

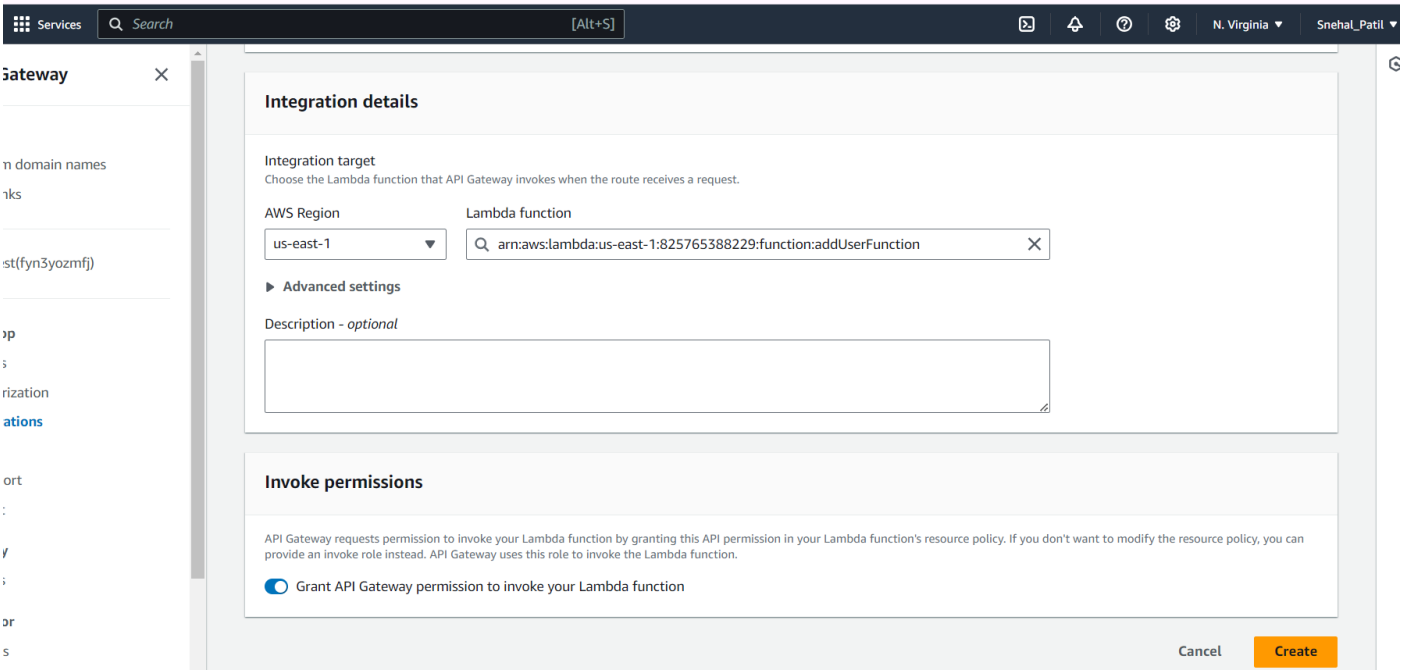
1. Inside API Gateway, click **Actions > Create Resource**.
2. Provide a **Resource Name** (e.g., user).
3. Keep the Resource Path as /user.
4. Click **Actions > Create Method**.
5. Select POST.



6. Under **Integration Type**, choose **Lambda Function**.

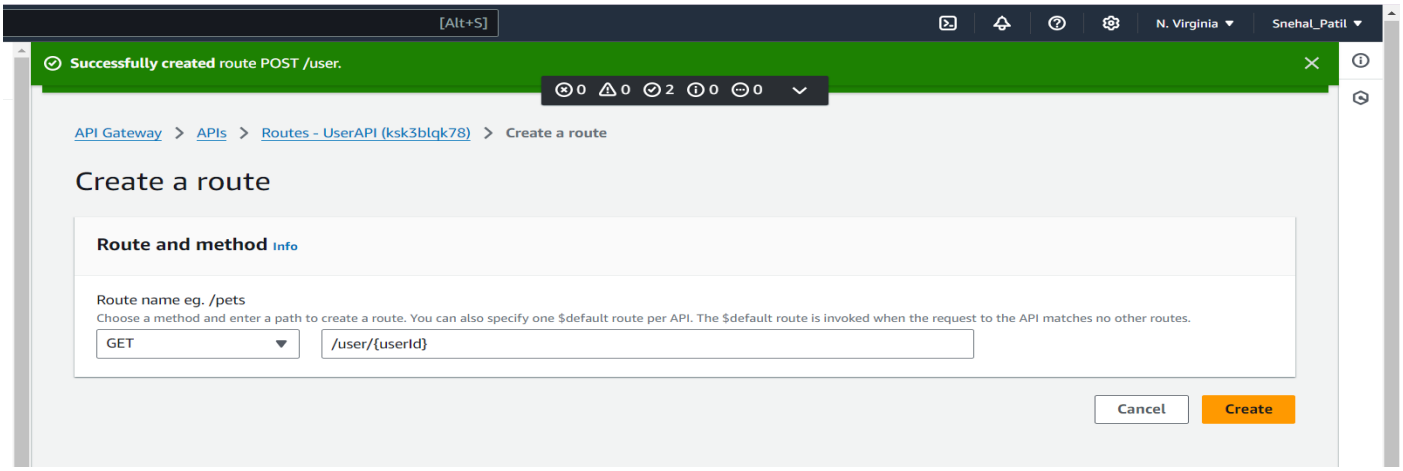


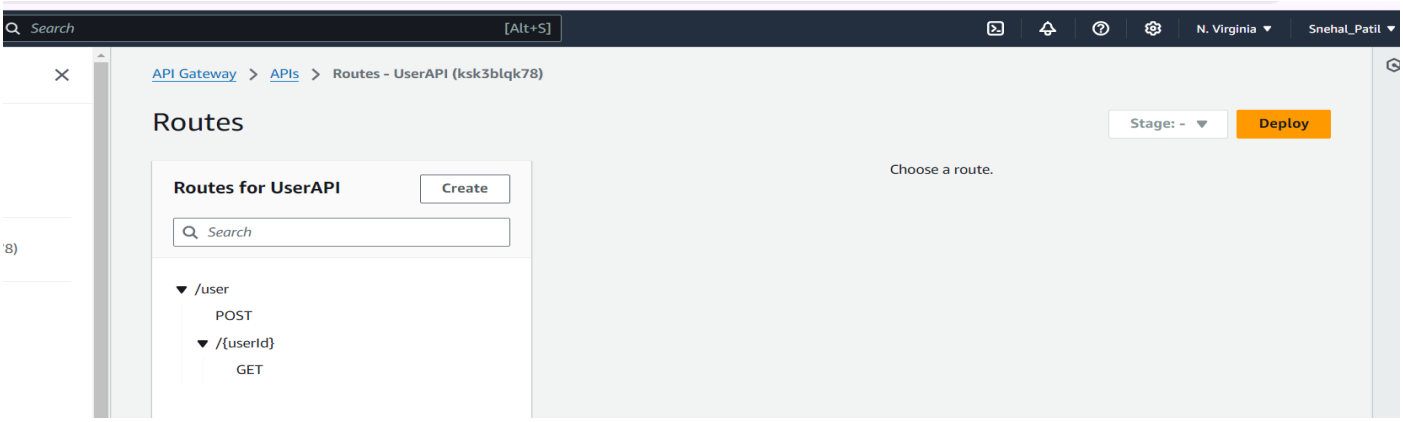
7. Select the addUserFunction Lambda function.
8. Click **create**.



Creating GET Method for Retrieving User by ID

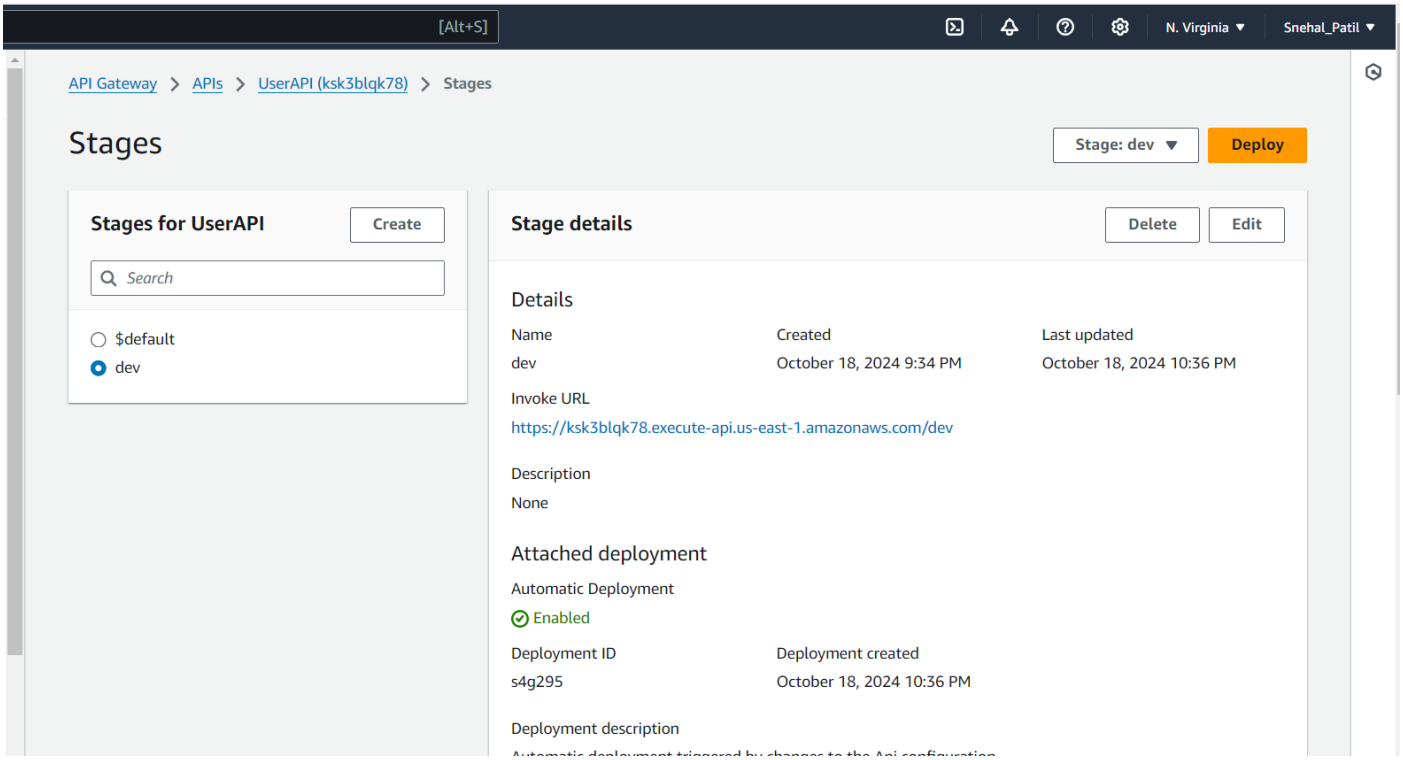
1. Under **Actions**, select **Create Method**.
2. Choose GET.
3. Under **Integration Type**, choose **Lambda Function**.
4. Select the getUserFunction Lambda function.
5. Set Path Parameters (e.g., /user/{userId}).
6. Click **Create**.



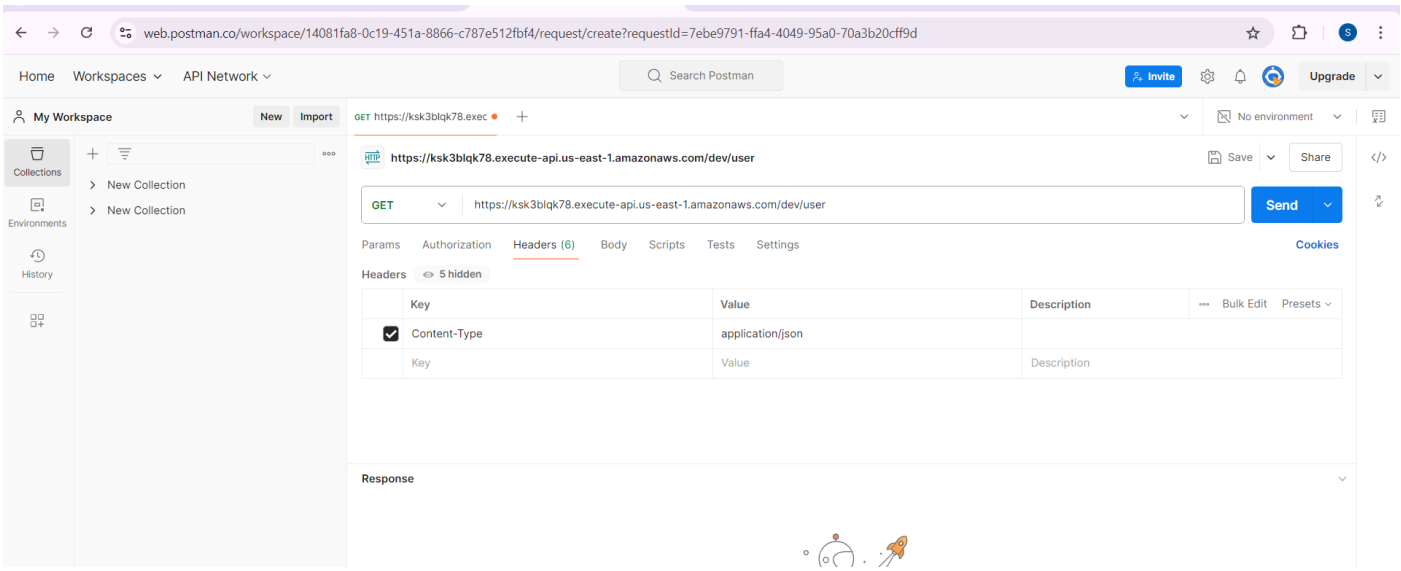


Deploying the API

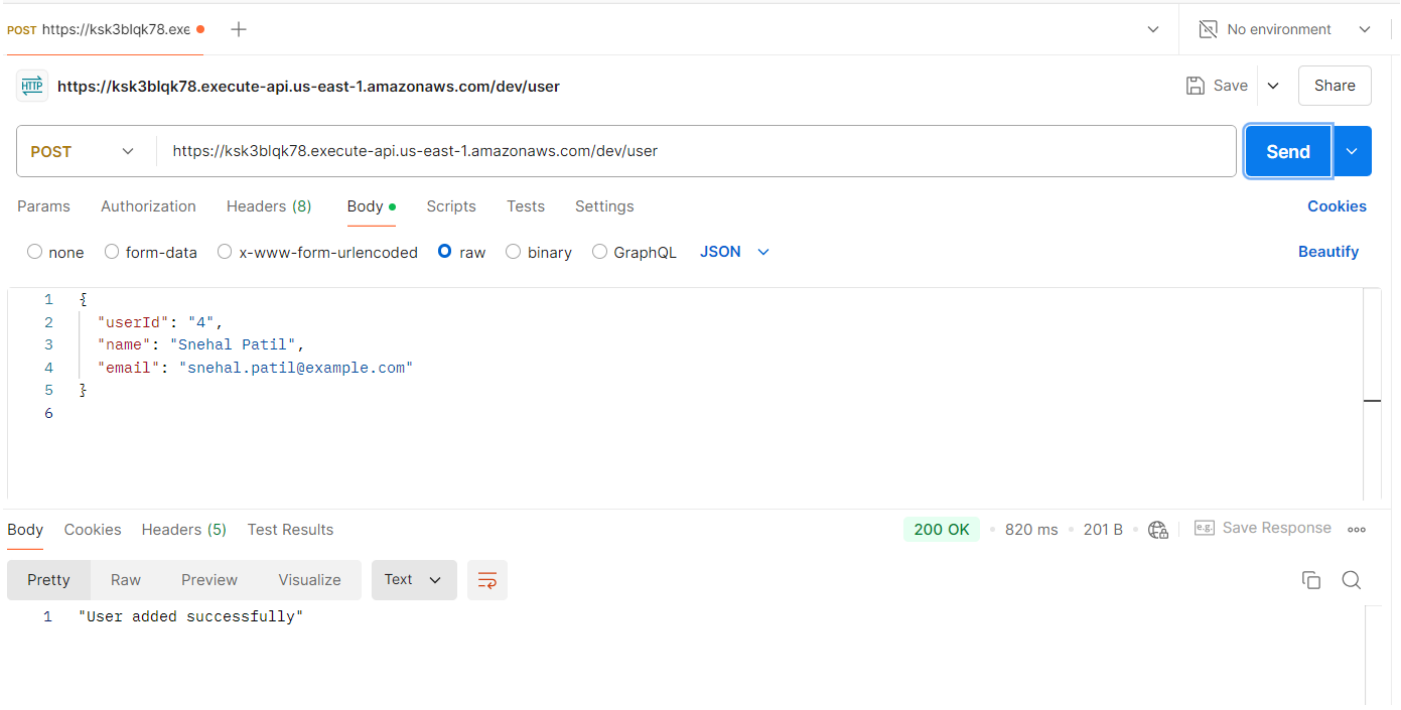
1. Click **Deploy API**.
2. Create a new deployment stage (e.g., dev).
3. You will receive the API endpoint URL once deployed.



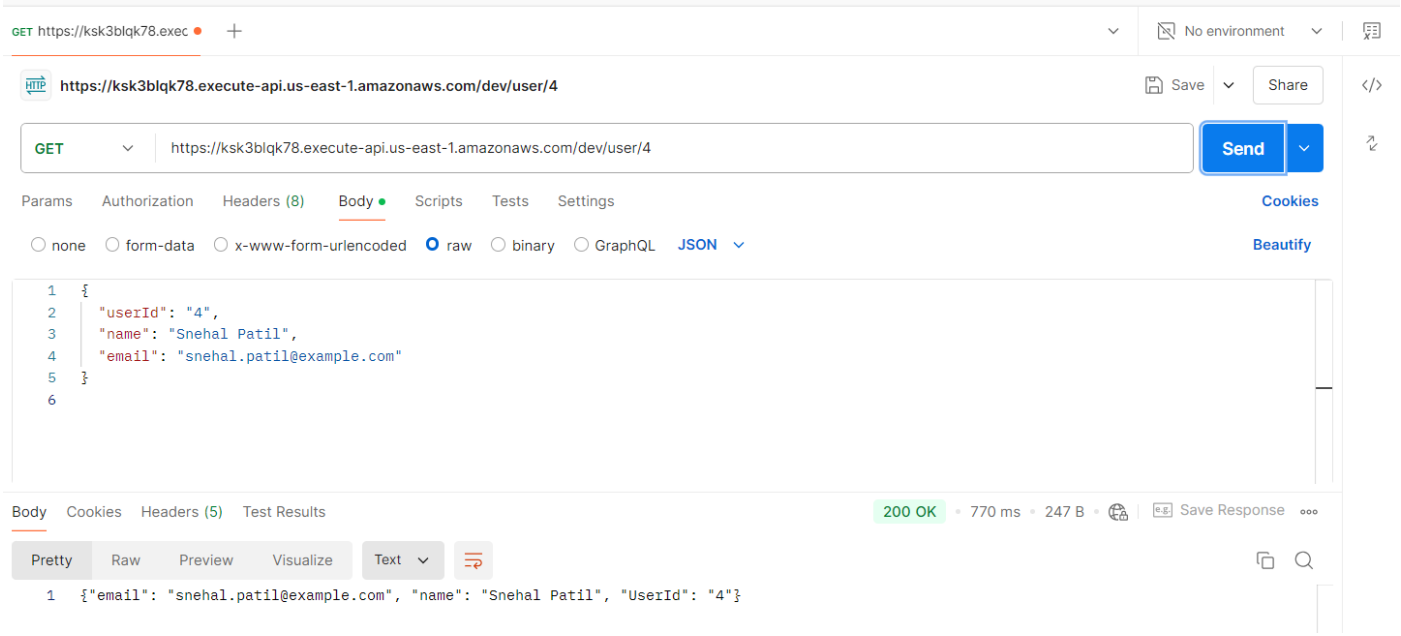
Testing the api on postman:



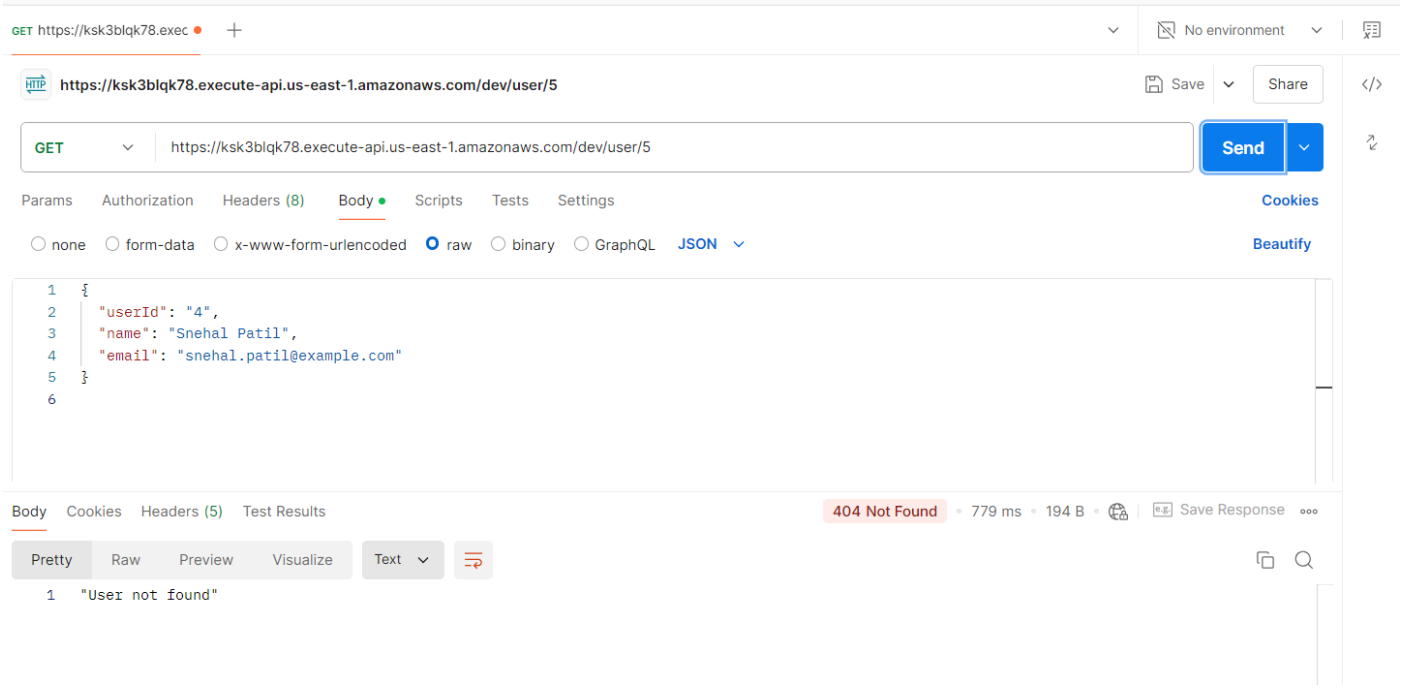
Post request to add data to the dynamo table



Get request to fetch the user data from table

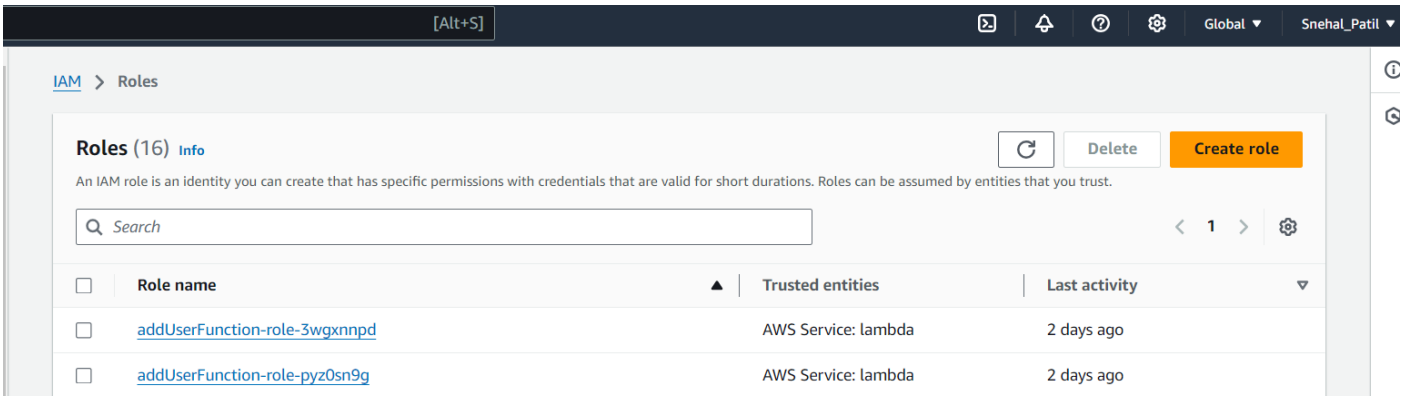


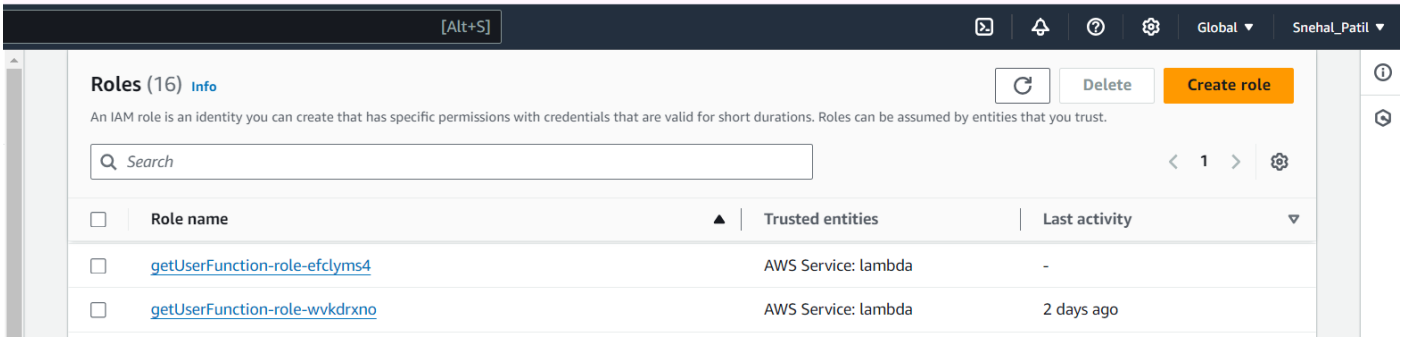
Trying to fetch the data not present in the table

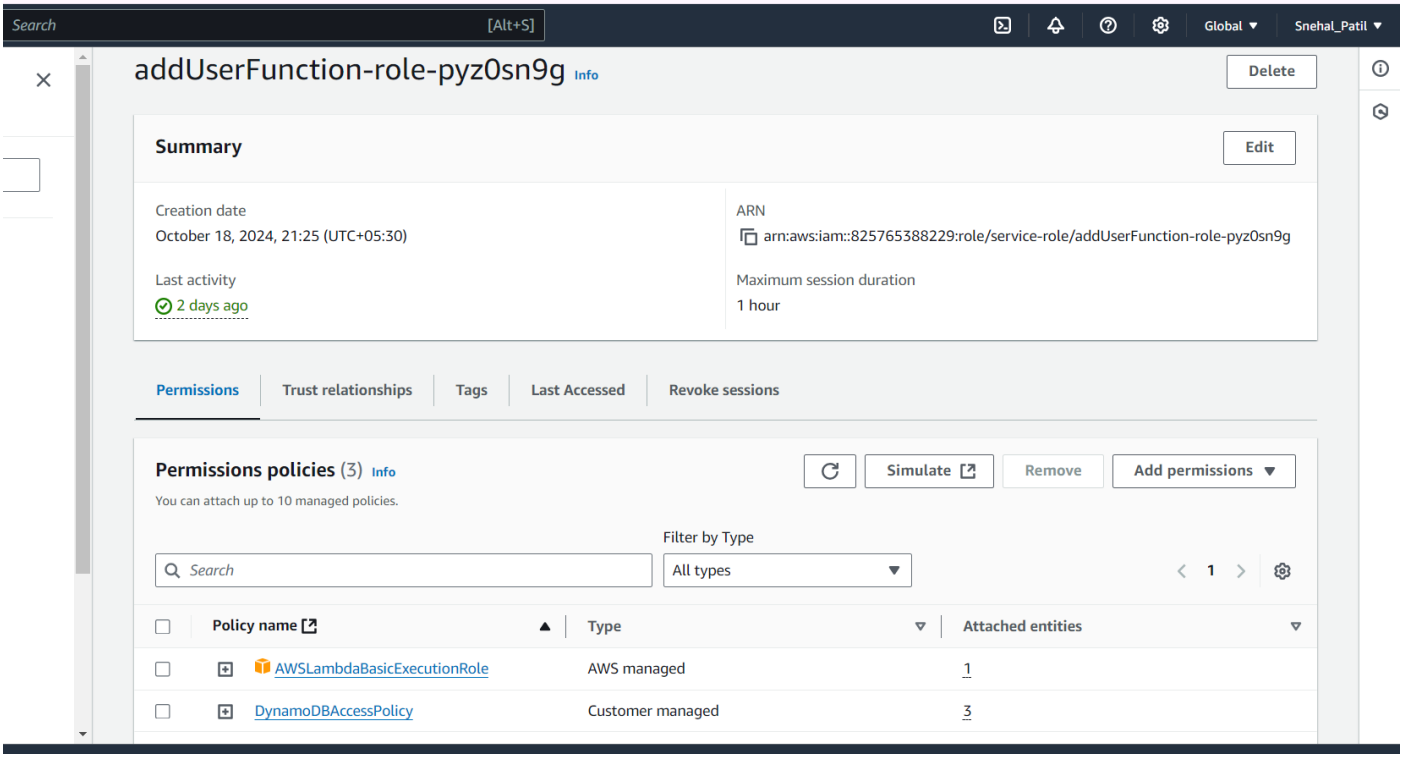


Guidelines

- Ensure IAM roles have correct permissions:
    - Attach DynamoDBAccessPolicy with actions: dynamodb:PutItem and dynamodb:Get Item.
    - Attach AWSLambdaBasicExecutionRole for logging.







Problems I Faced During execution :

Problem 1: Permissions Error

- **Issue**: Encountered AccessDeniedException when the Lambda function tried to perform op erations 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: DynamoDB Validation Error

- **Issue**: Encountered ValidationException due to missing UserId key in the item.
- **Solution**: Ensured the UserId key in the item matched the DynamoDB table's partition key:

javascript

Copy

```
const params = {
    TableName: 'Users',
    Item: {
        'UserId': userId,
        'name': name,
        'email': 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 nee d 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 config uration and diligent cost management.

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