**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.

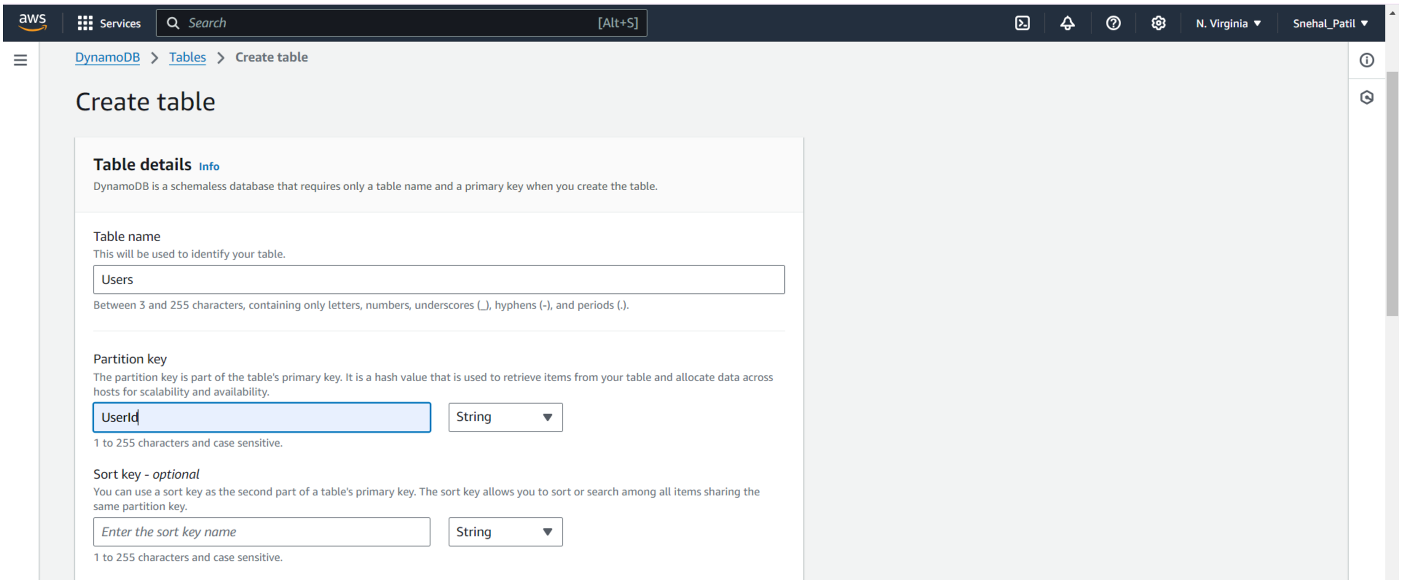
**4. Managing Salon Services and Products**

- **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**.



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

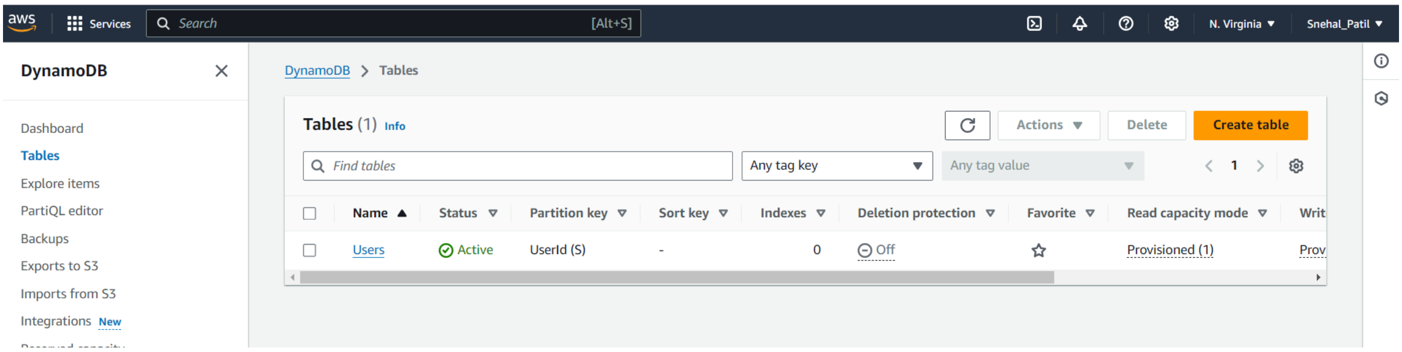
1 to 255 characters and case sensitive.

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.



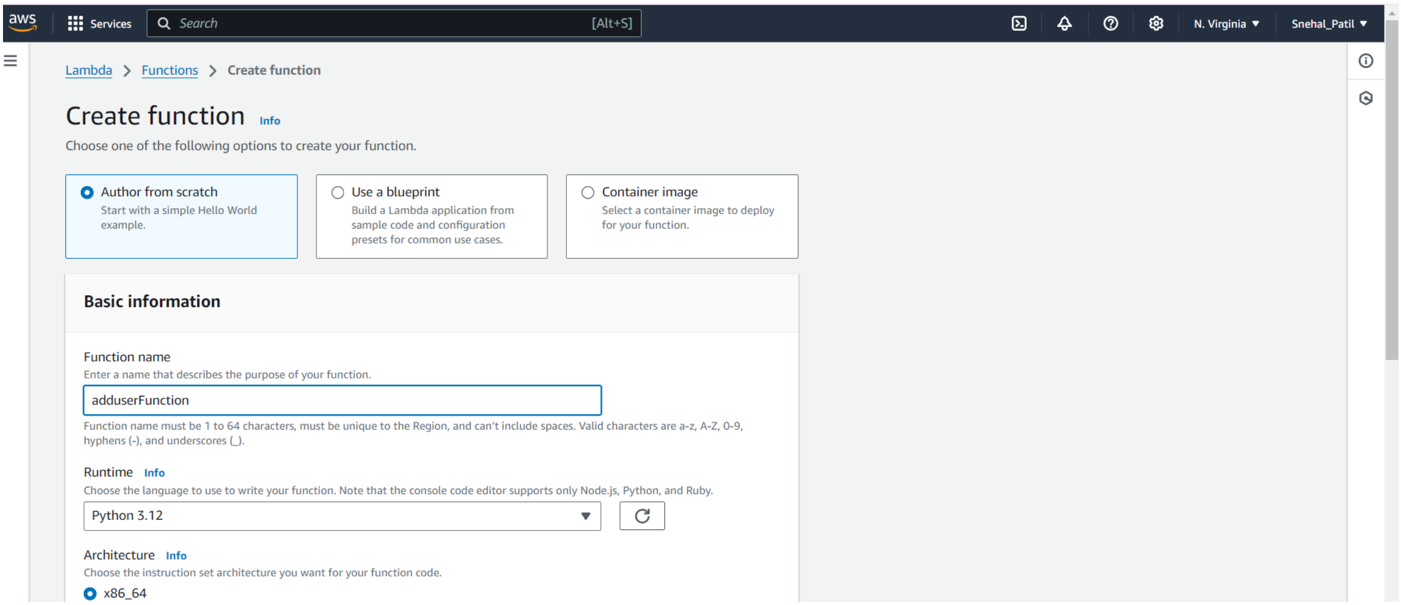
Tables (1) Info

| □ | Name ▲ | Status | Partition key | Sort key | Indexes | Deletion protection | Favorite | Read capacity mode | Writ |
|---|---|---|---|---|---|---|---|---|---|
| □ | Users | ✓ Active | UserId (S) | - | 0 | ⊖ Off | ☆ | Provisioned (1) | Prov |

### 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



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.

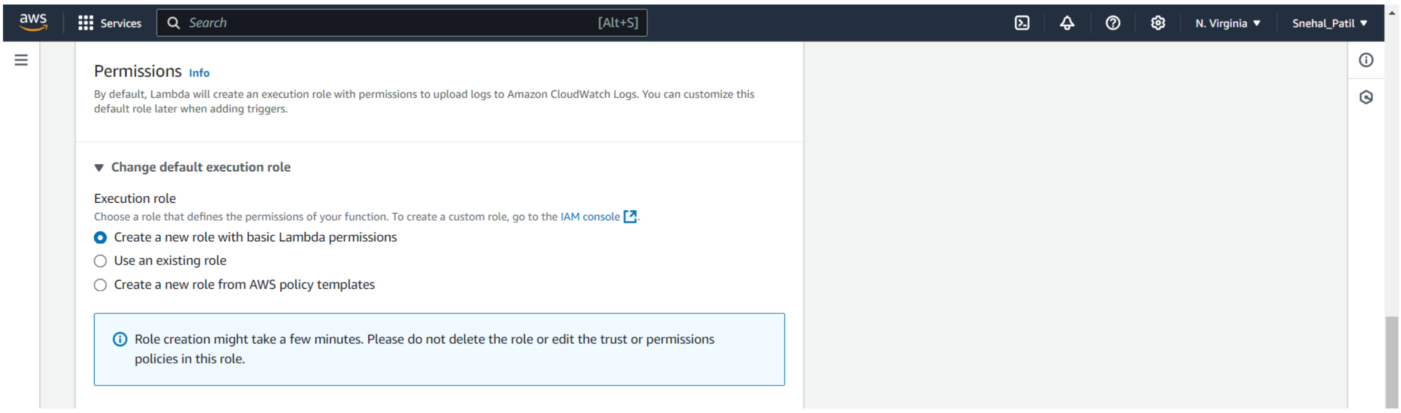
Basic information

Function name
Enter a name that describes the purpose of your function.
adduserFunction
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.
Python 3.12

Architecture Info
Choose the instruction set architecture you want for your function code.
● x86_64

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



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.

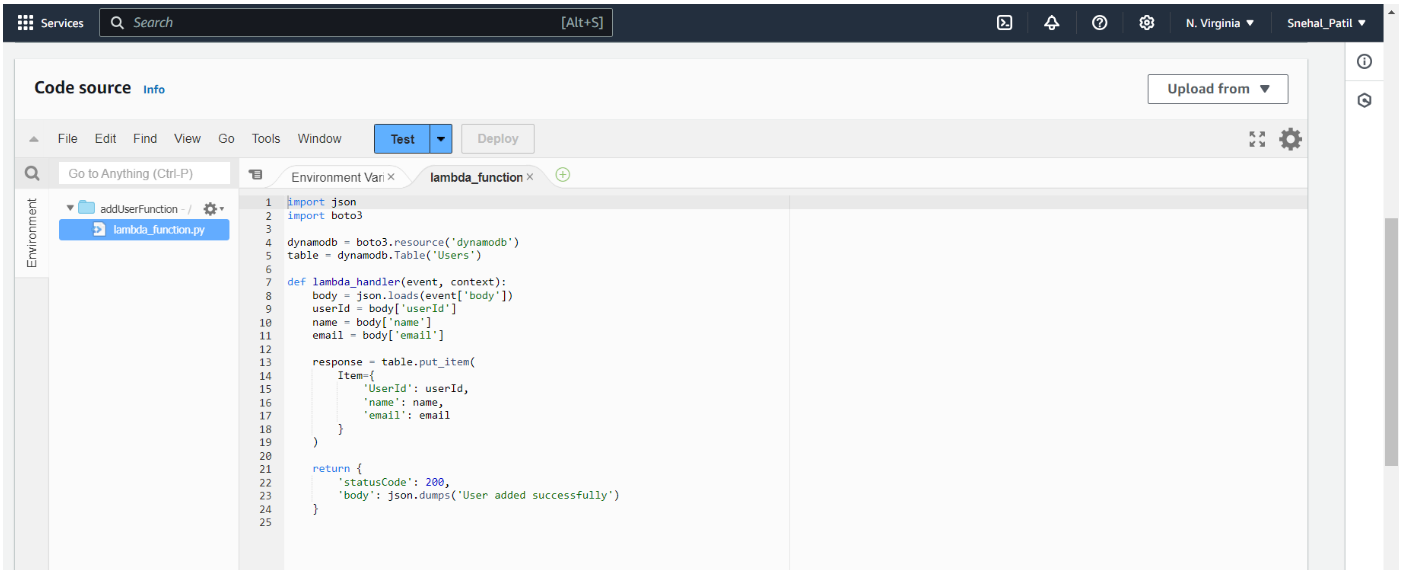
▼ Change default execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

● Create a new role with basic Lambda permissions
○ Use an existing role
○ Create a new role from AWS policy templates

ⓘ Role creation might take a few minutes. Please do not delete the role or edit the trust or permissions policies in this role.

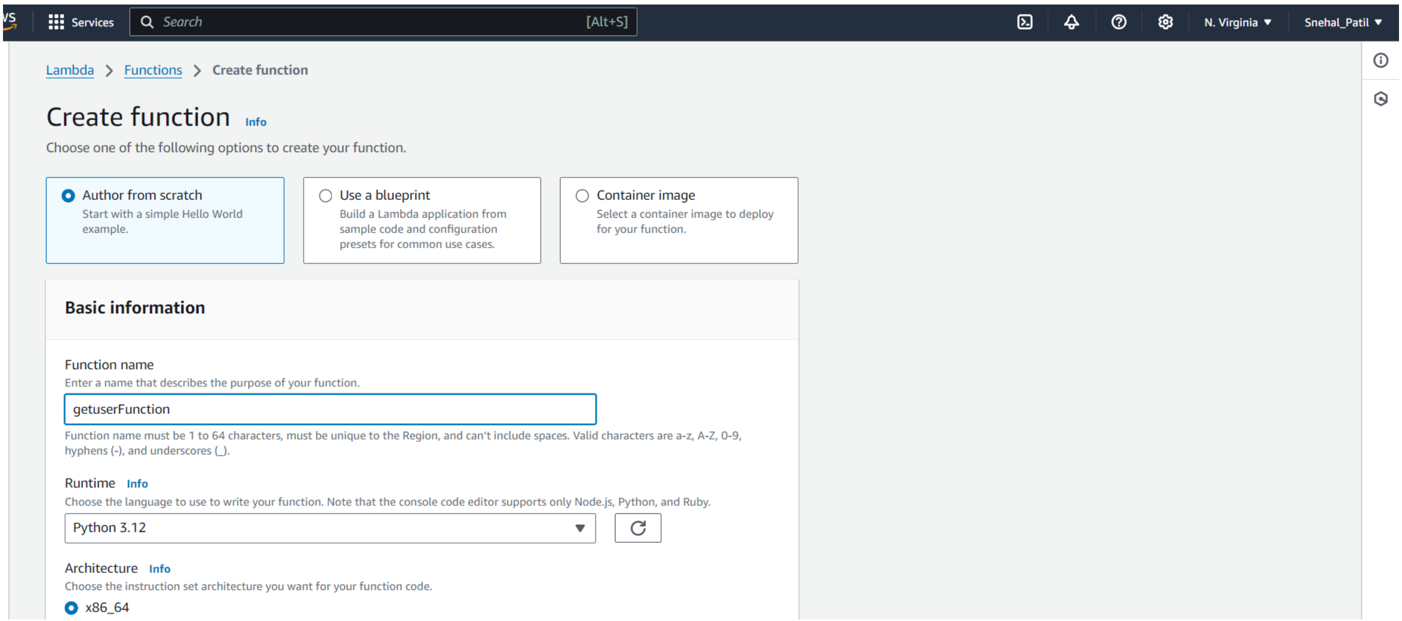
**Code for** addUserFunction:



```python
import json
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Users')

def lambda_handler(event, context):
    body = json.loads(event['body'])
    userId = body['userId']
    name = body['name']
    email = body['email']

    response = table.put_item(
        Item={
            'UserId': userId,
            'name': name,
            'email': email
        }
    )

    return {
        'statusCode': 200,
        'body': json.dumps('User added successfully')
    }
```

## 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.



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.

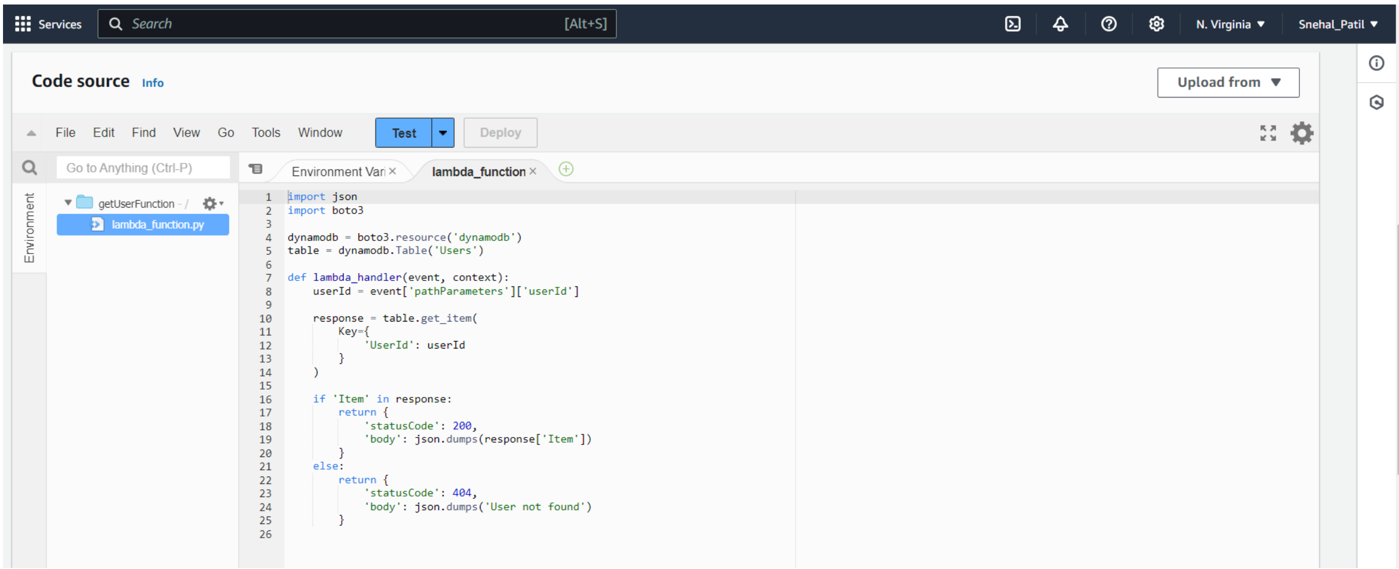
Basic information

Function name
Enter a name that describes the purpose of your function.
getuserFunction
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.
Python 3.12

Architecture Info
Choose the instruction set architecture you want for your function code.
● x86_64

## Code for getUserFunction:

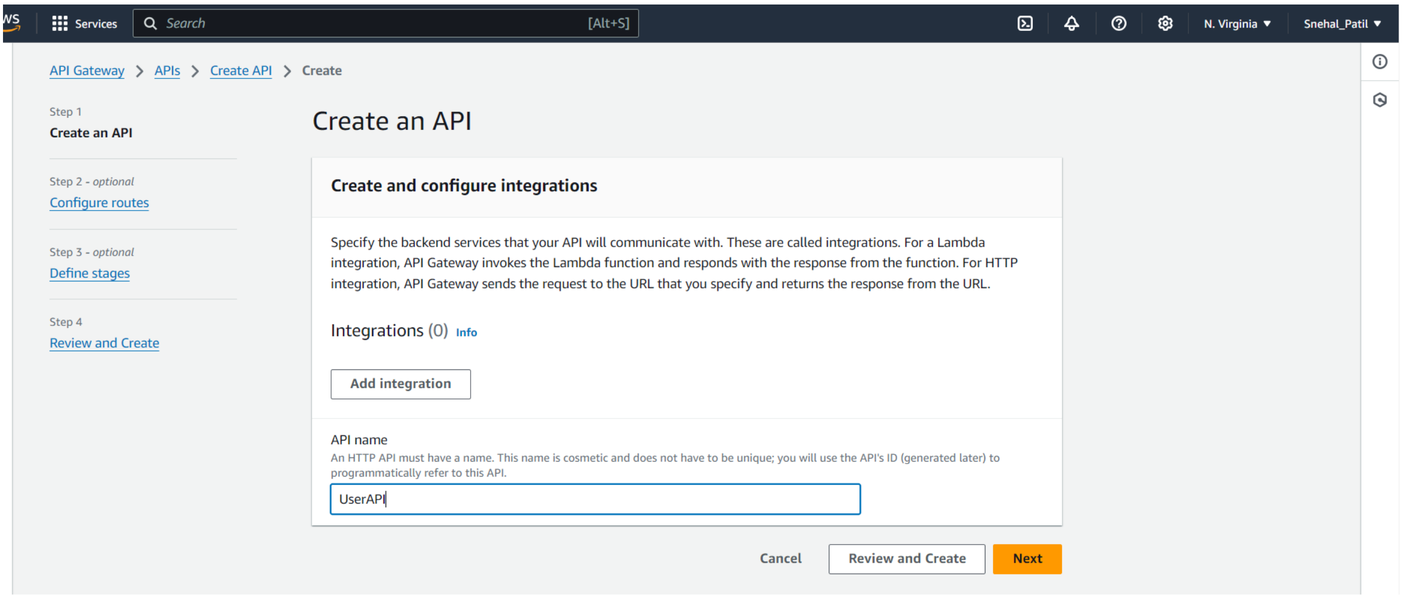


```python
import json
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Users')

def lambda_handler(event, context):
    userId = event['pathParameters']['userId']

    response = table.get_item(
        Key={
            'UserId': userId
        }
    )

    if 'Item' in response:
        return {
            'statusCode': 200,
            'body': json.dumps(response['Item'])
        }
    else:
        return {
            'statusCode': 404,
            'body': json.dumps('User not found')
        }

```

## 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).



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

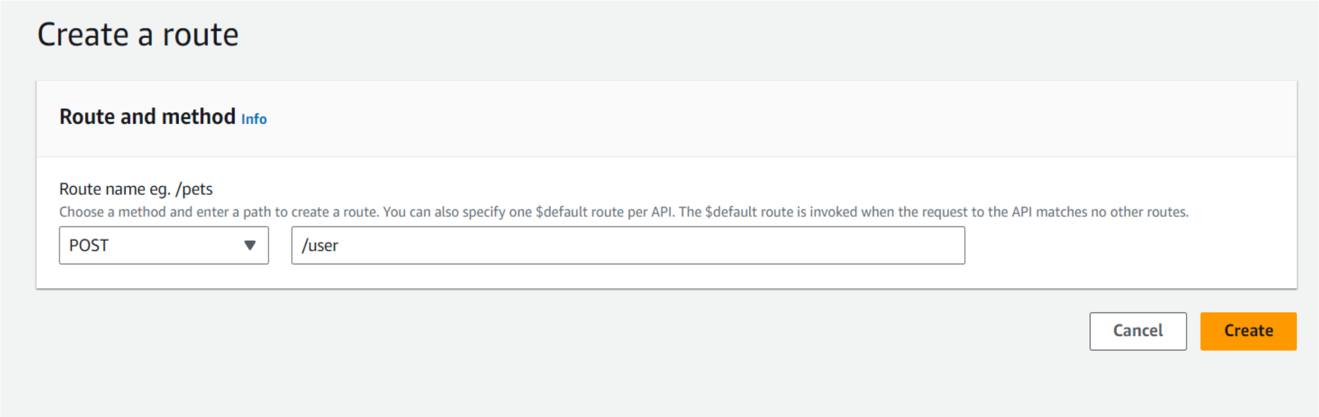
API name

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.

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.



Create a route

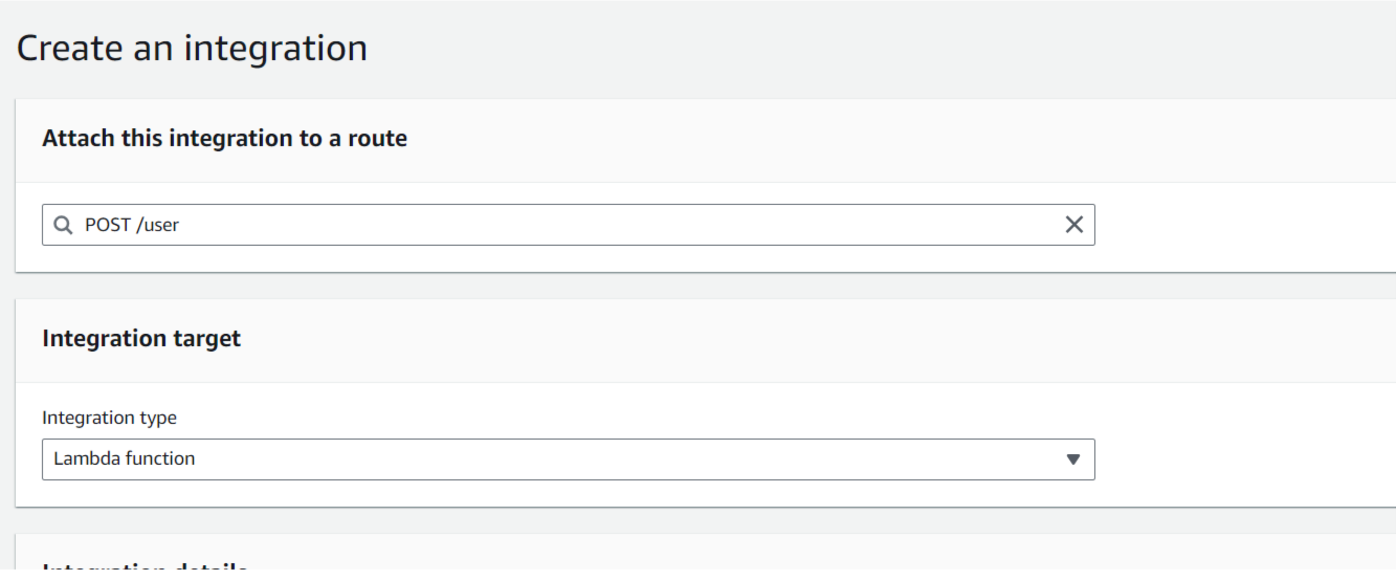
Route and method Info

Route name eg. /pets

Choose a method and enter a path to create a route. You can also specify one $default route per API. The $default route is invoked when the request to the API matches no other routes.

POST ▼ /user

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



Create an integration

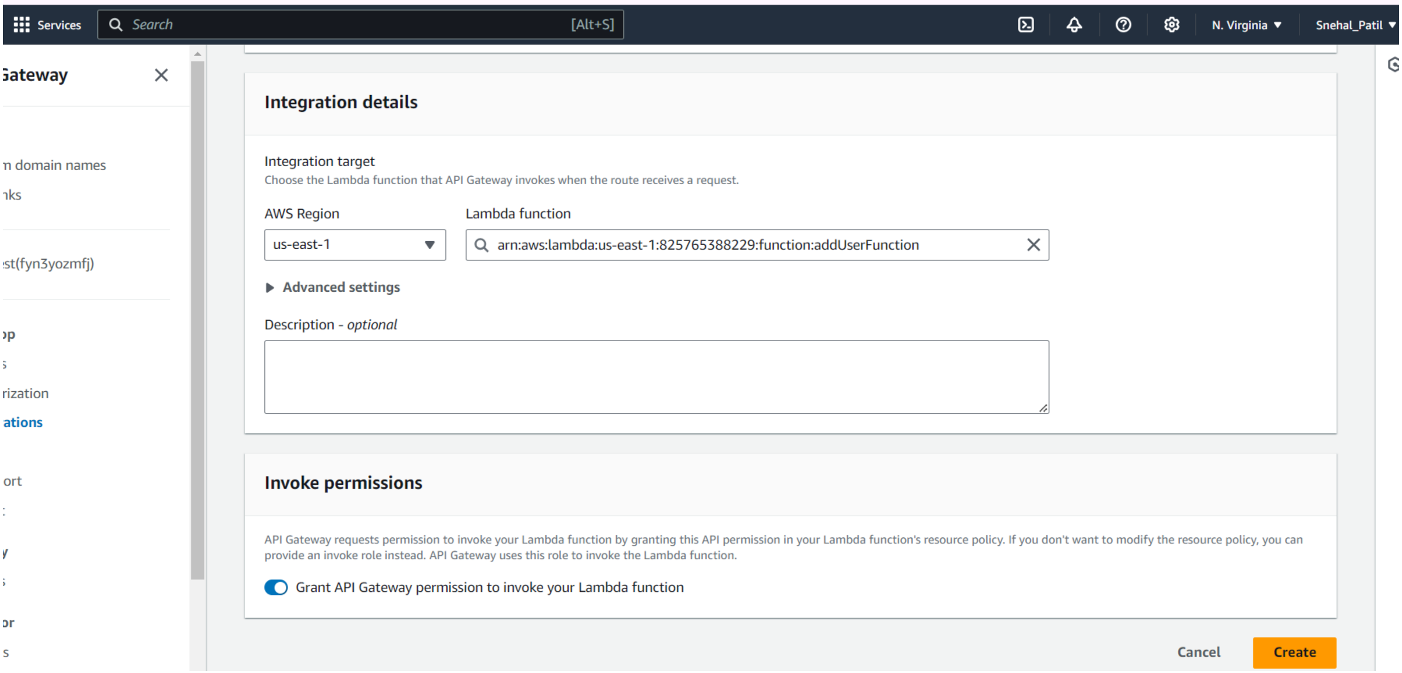
Attach this integration to a route

POST /user

Integration target

Integration type: Lambda function

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



Integration details

Integration target

Choose the Lambda function that API Gateway invokes when the route receives a request.

AWS Region: us-east-1

Lambda function: arn:aws:lambda:us-east-1:825765388229:function:addUserFunction

▶ Advanced settings

Description - *optional*

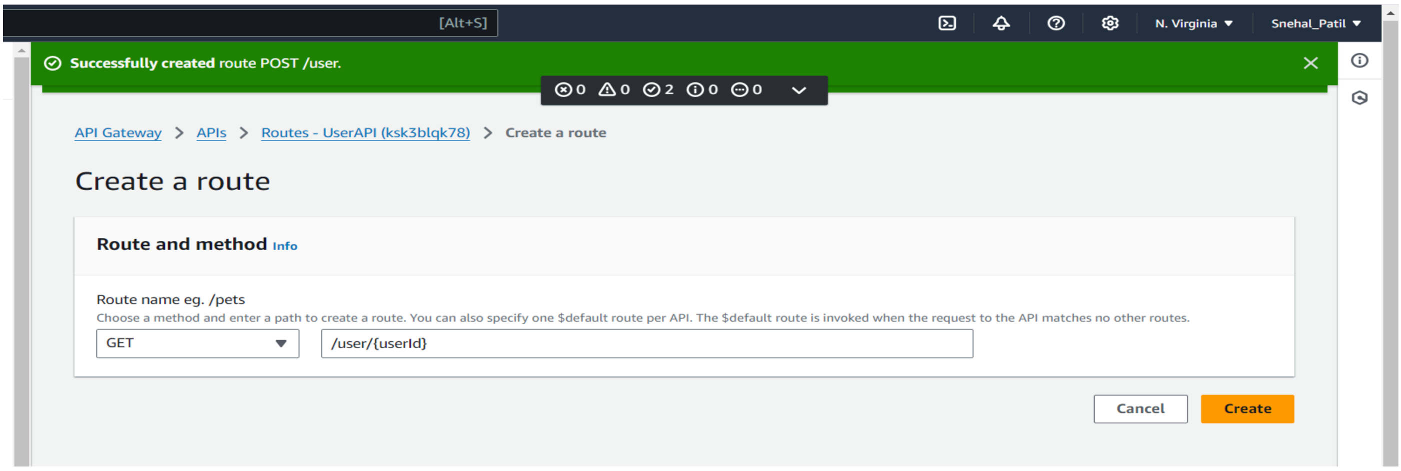
Invoke permissions

API Gateway requests permission to invoke your Lambda function by granting this API permission in your Lambda function's resource policy. If you don't want to modify the resource policy, you can provide an invoke role instead. API Gateway uses this role to invoke the Lambda function.

● Grant API Gateway permission to invoke your Lambda function

## 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**.



✓ **Successfully created** route POST /user.

API Gateway > APIs > Routes - UserAPI (ksk3blqk78) > Create a route

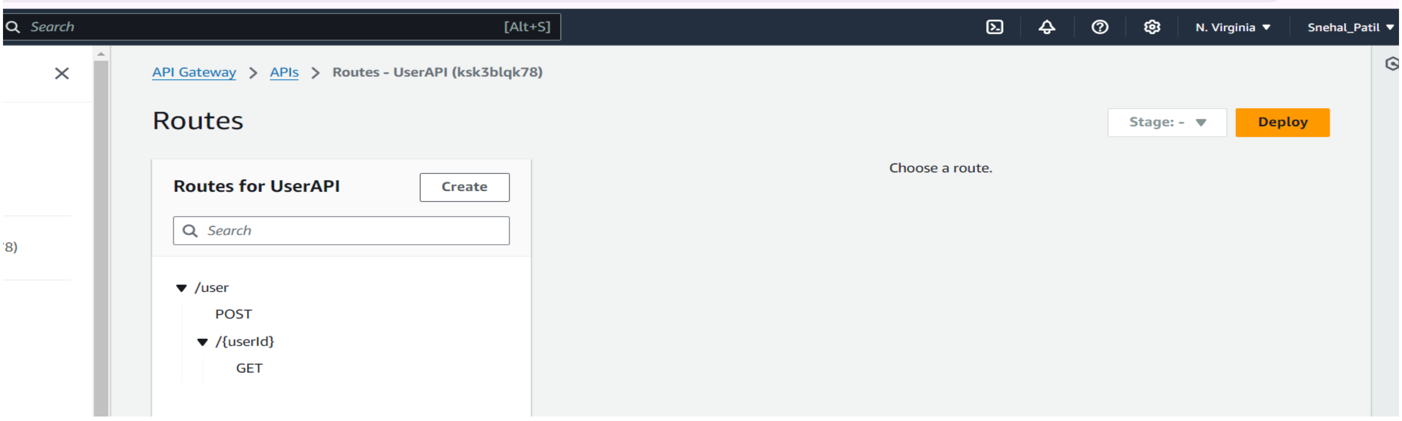
Create a route

**Route and method** Info

Route name eg. /pets

Choose a method and enter a path to create a route. You can also specify one $default route per API. The $default route is invoked when the request to the API matches no other routes.

GET /user/{userId}



API Gateway > APIs > Routes - UserAPI (ksk3blqk78)

Routes

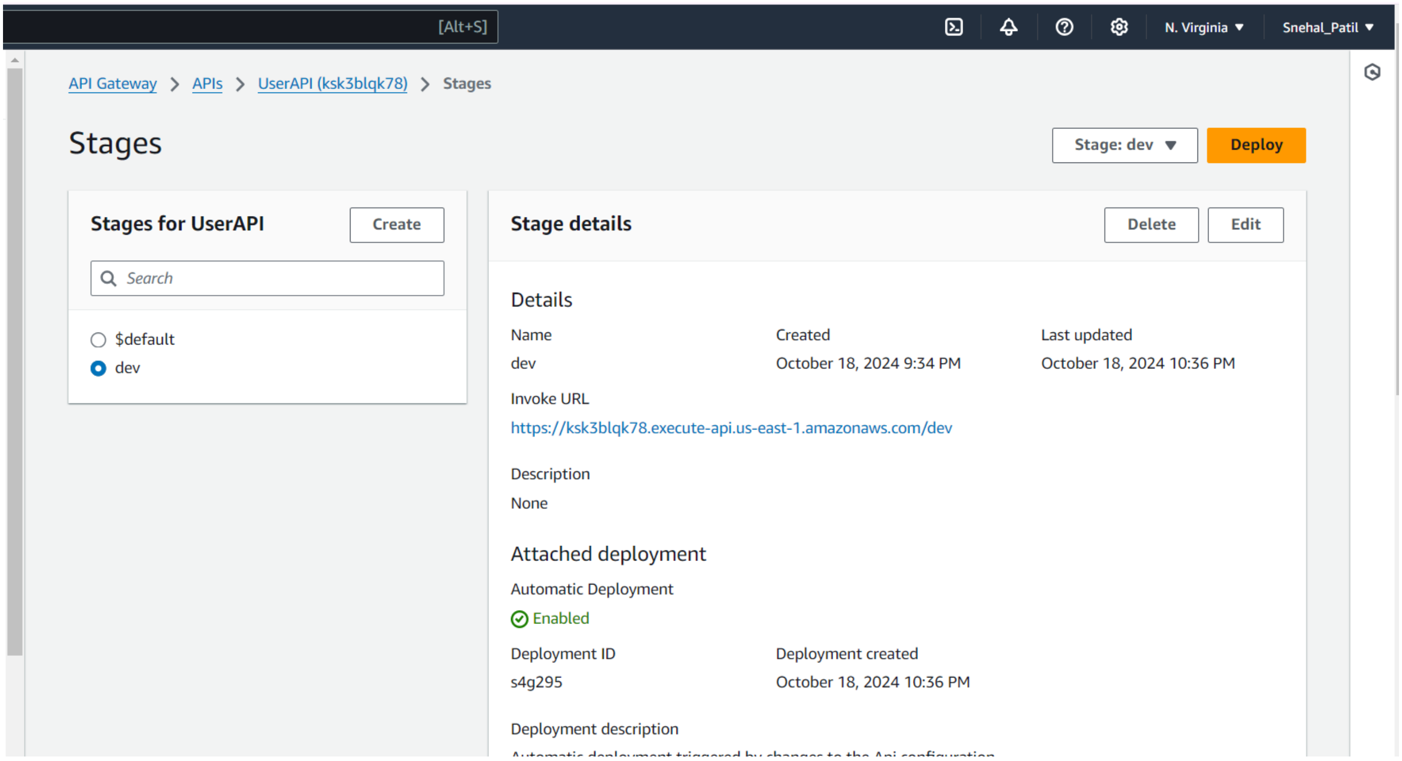
Routes for UserAPI

Choose a route.

- /user
  - POST
  - /{userId}
    - GET

## 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.



API Gateway > APIs > UserAPI (ksk3blqk78) > Stages

Stages

Stages for UserAPI

○ $default
● dev

Stage details

Details

| Name | Created | Last updated |
|---|---|---|
| dev | October 18, 2024 9:34 PM | October 18, 2024 10:36 PM |

Invoke URL

https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev

Description

None

Attached deployment

Automatic Deployment

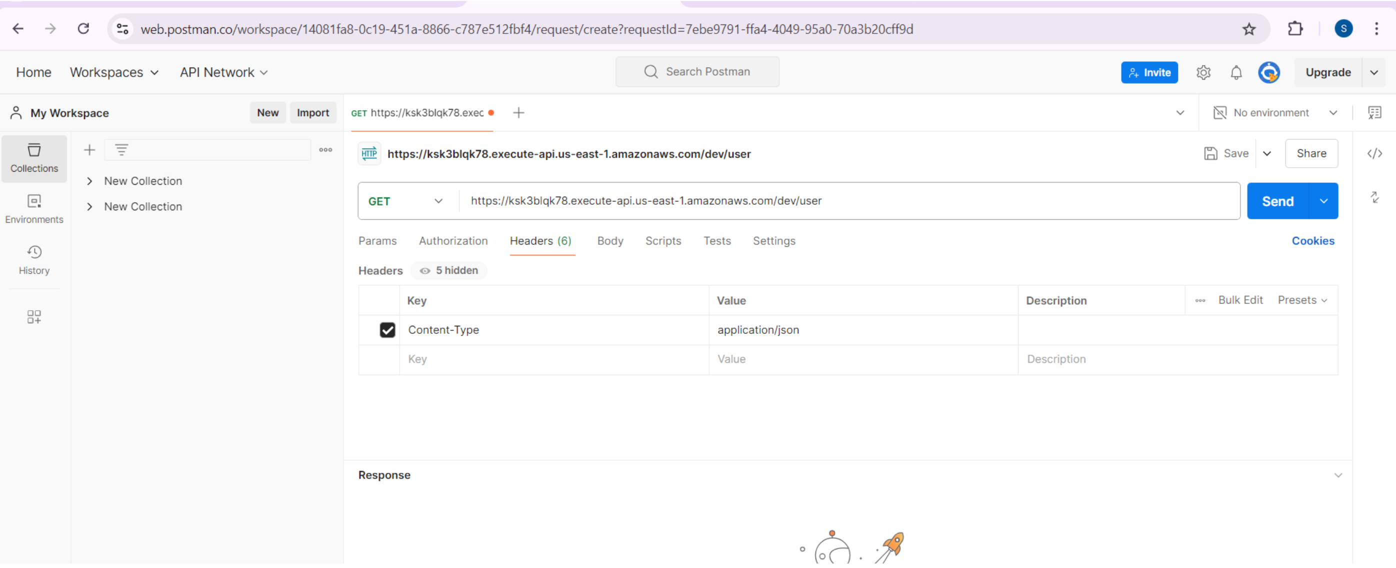
✓ Enabled

| Deployment ID | Deployment created |
|---|---|
| s4g295 | October 18, 2024 10:36 PM |

Deployment description

Automatic deployment triggered by changes to the Api configuration

Testing the api on postman:



web.postman.co/workspace/14081fa8-0c19-451a-8866-c787e512fbf4/request/create?requestId=7ebe9791-ffa4-4049-95a0-70a3b20cff9d

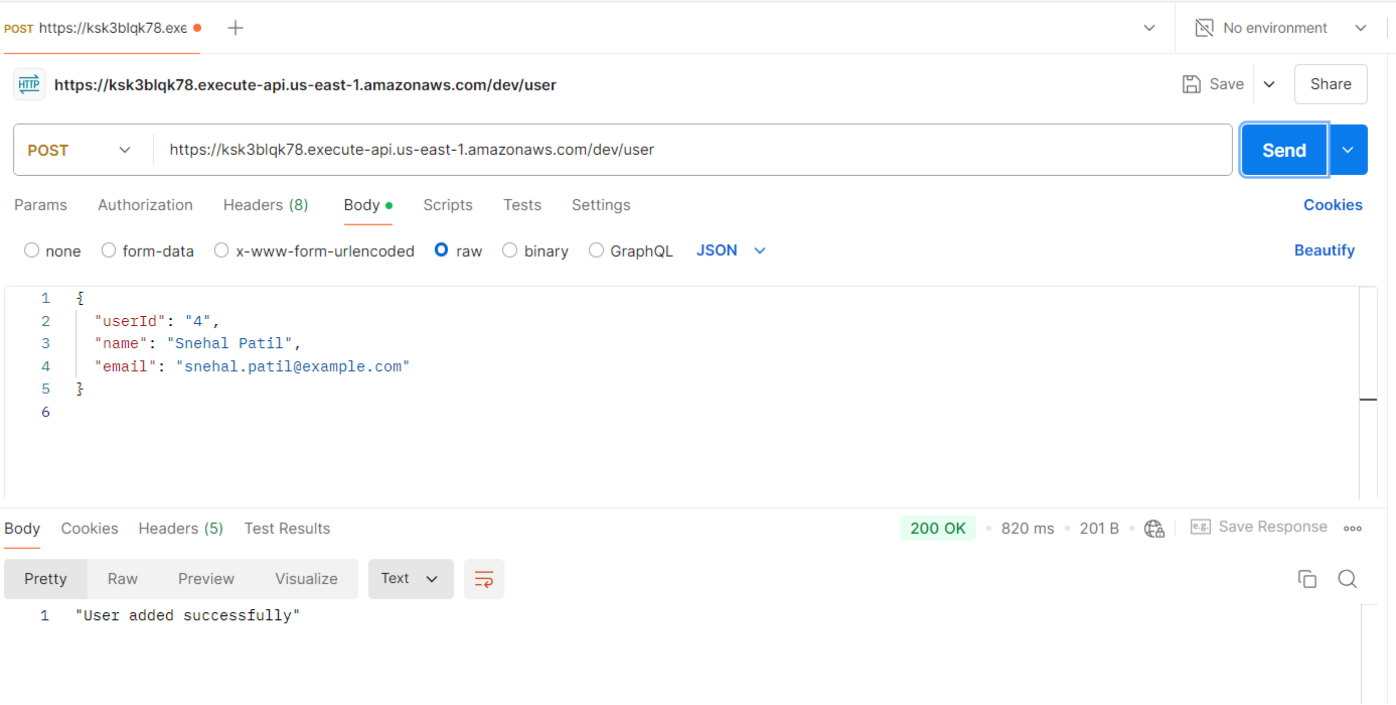
https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

GET https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

Headers 5 hidden

| | Key | Value | Description |
|---|---|---|---|
| ☑ | Content-Type | application/json | |
| | Key | Value | Description |

Post request to add data to the dynamo table



https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

POST https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

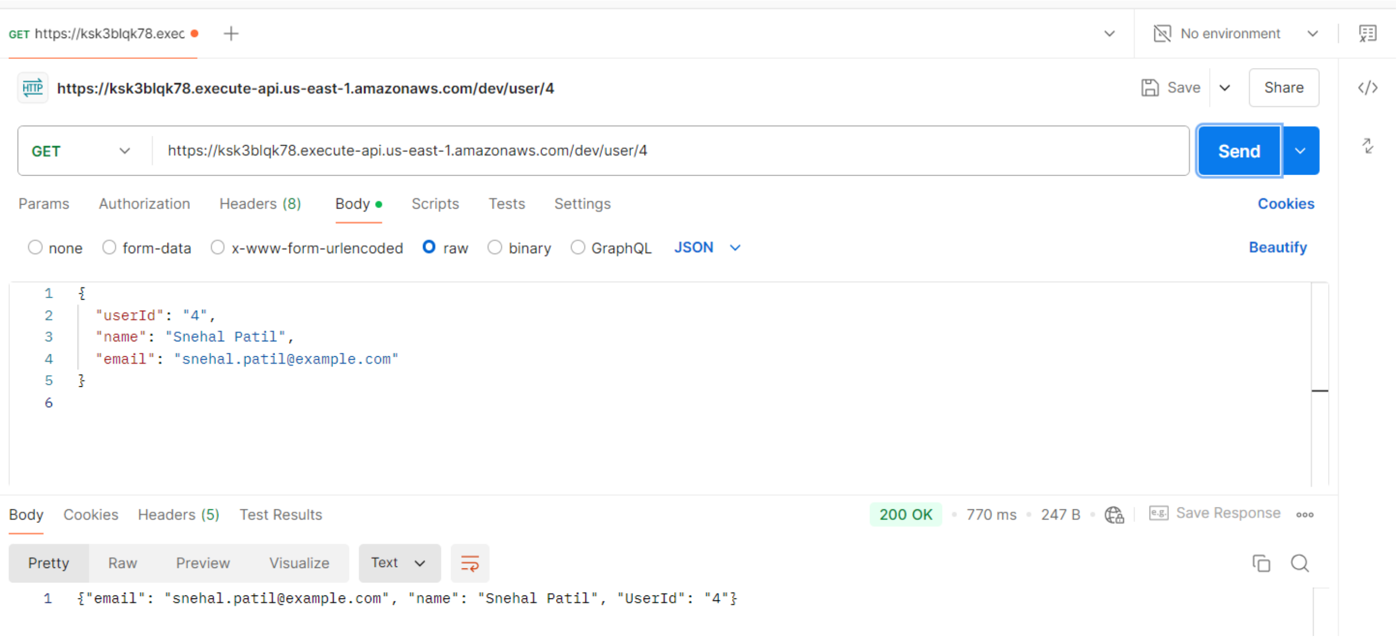
○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
  "userId": "4",
  "name": "Snehal Patil",
  "email": "snehal.patil@example.com"
}
```

Body 200 OK 820 ms 201 B

```
"User added successfully"
```

## Get request to fetch the user data from table



https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user/4

GET https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user/4

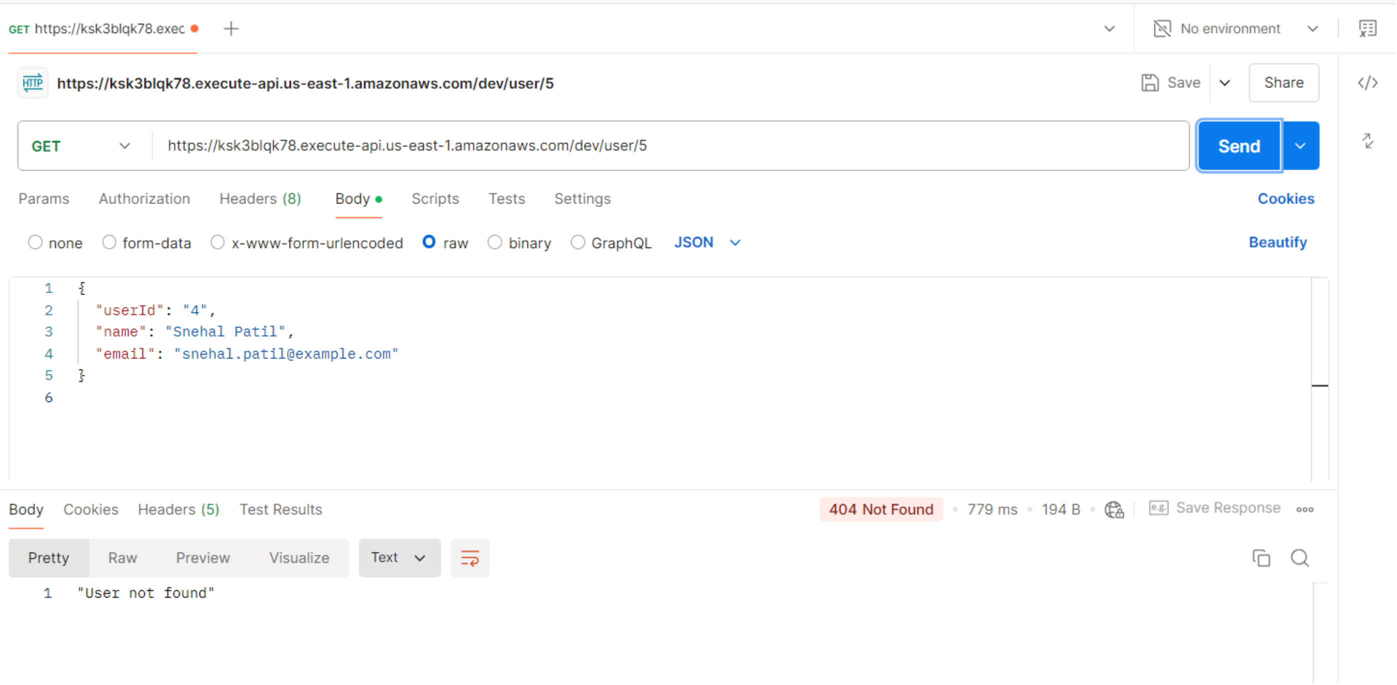
○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
  "userId": "4",
  "name": "Snehal Patil",
  "email": "snehal.patil@example.com"
}
```

200 OK • 770 ms • 247 B

```
{"email": "snehal.patil@example.com", "name": "Snehal Patil", "UserId": "4"}
```

## Trying to fetch the data not present in the table



https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user/5

GET https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user/5

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
  "userId": "4",
  "name": "Snehal Patil",
  "email": "snehal.patil@example.com"
}
```

404 Not Found • 779 ms • 194 B

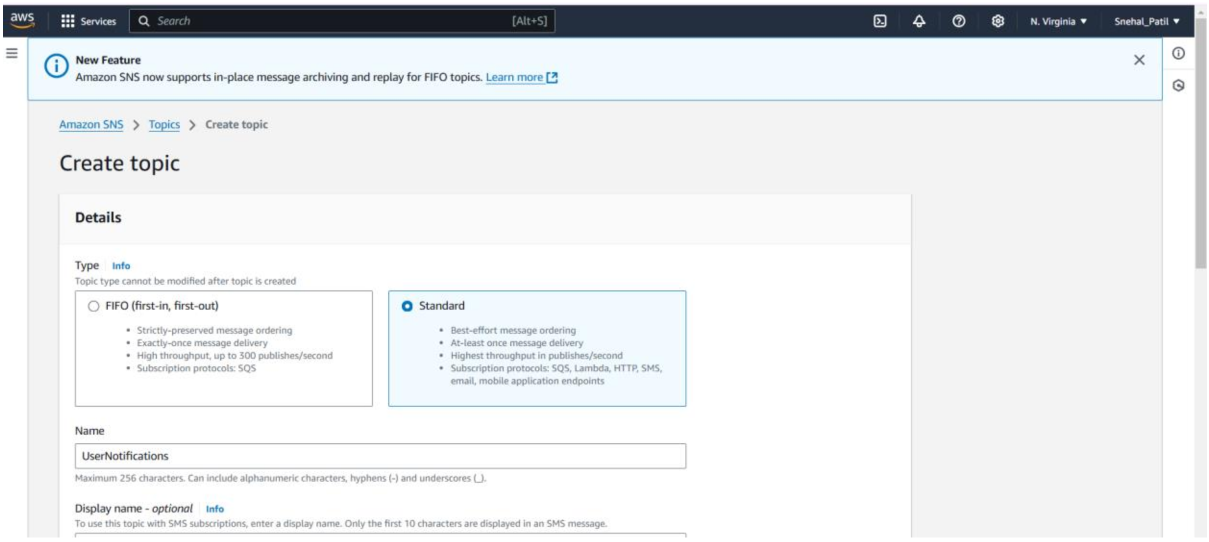
```
"User not found"
```

Extra:

**Steps for Implementing AWS SNS in Serverless API**

## 1. Set Up AWS SNS

1. **Create an SNS Topic**:
    - Go to the **[AWS SNS Console](#)**.
    - Click **Create Topic**.
    - Choose a topic type (Standard or FIFO).
    - Enter a topic name (e.g., UserNotifications).
    - Click **Create Topic** and save the ARN provided.



**New Feature**
Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more

Amazon SNS > Topics > Create topic

Create topic

**Details**

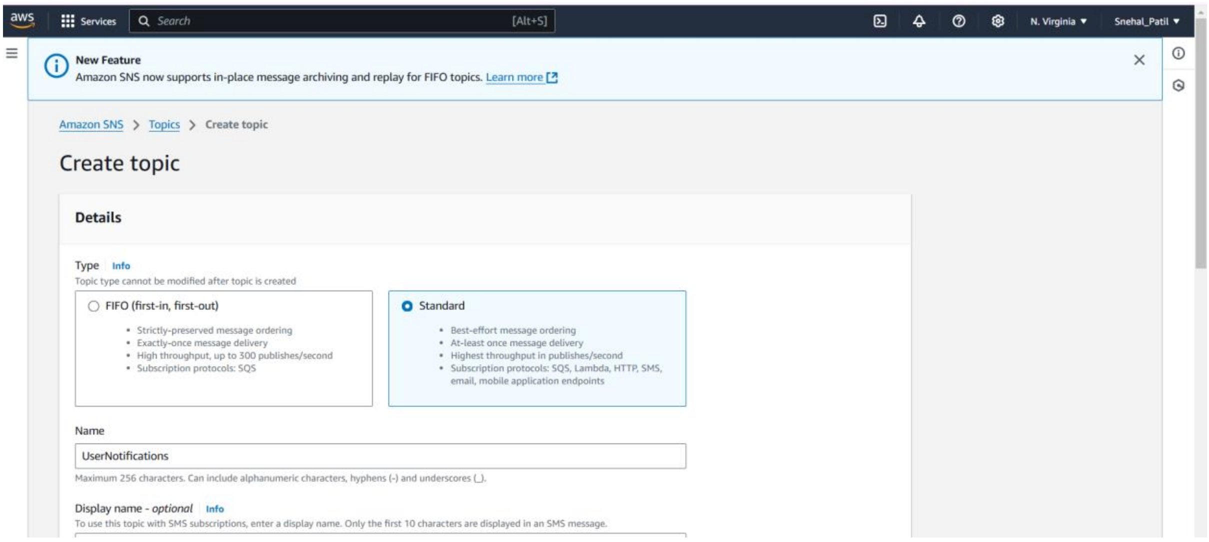
Type Info
Topic type cannot be modified after topic is created

○ FIFO (first-in, first-out)
- Strictly-preserved message ordering
- Exactly-once message delivery
- High throughput, up to 300 publishes/second
- Subscription protocols: SQS

● Standard
- Best-effort message ordering
- At-least once message delivery
- Highest throughput in publishes/second
- Subscription protocols: SQS, Lambda, HTTP, SMS, email, mobile application endpoints

Name
UserNotifications
Maximum 256 characters. Can include alphanumeric characters, hyphens (-) and underscores (_).

Display name - *optional* Info
To use this topic with SMS subscriptions, enter a display name. Only the first 10 characters are displayed in an SMS message.


**New Feature**
Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more

Amazon SNS > Topics > Create topic

Create topic

**Details**

Type Info
Topic type cannot be modified after topic is created

○ FIFO (first-in, first-out)
- Strictly-preserved message ordering
- Exactly-once message delivery
- High throughput, up to 300 publishes/second
- Subscription protocols: SQS

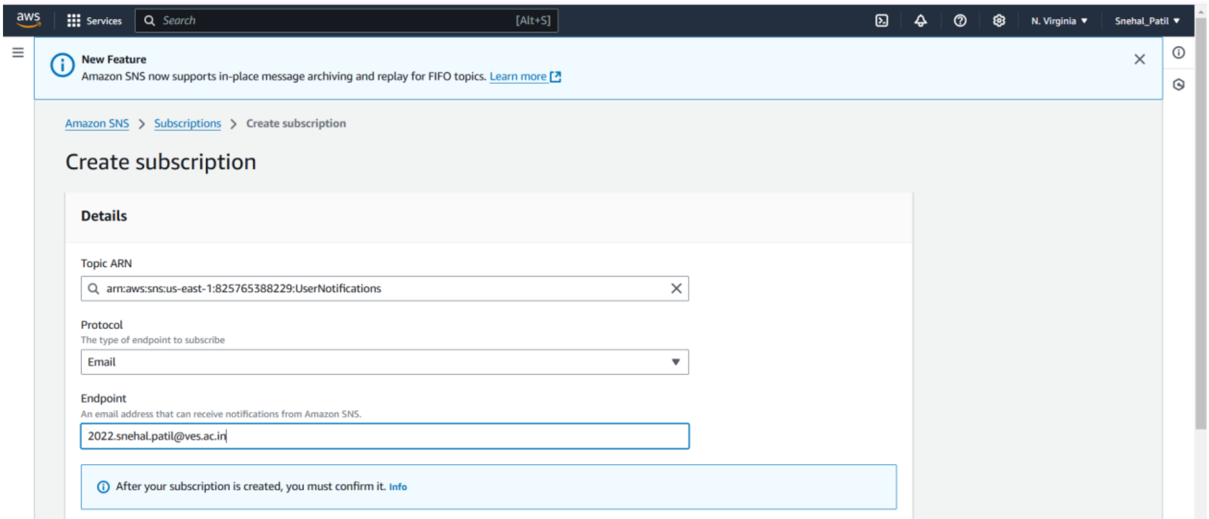
● Standard
- Best-effort message ordering
- At-least once message delivery
- Highest throughput in publishes/second
- Subscription protocols: SQS, Lambda, HTTP, SMS, email, mobile application endpoints

Name
UserNotifications
Maximum 256 characters. Can include alphanumeric characters, hyphens (-) and underscores (_).

Display name - *optional* Info
To use this topic with SMS subscriptions, enter a display name. Only the first 10 characters are displayed in an SMS message.

2. **Subscribe to the SNS Topic:**

- Open the topic you just created.
- Click Create Subscription.
- Choose a protocol (e.g., Email, SMS).
- Enter the endpoint (e.g., email address or phone number).
- Click Create Subscription.



New Feature

Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more

Amazon SNS > Subscriptions > Create subscription

Create subscription

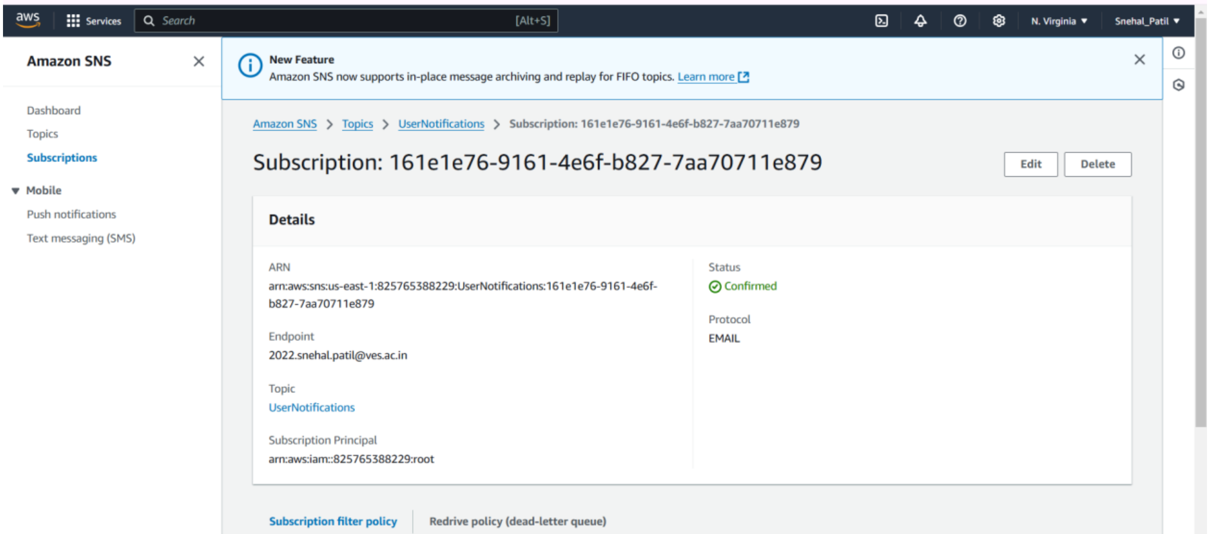
Details

Topic ARN: arn:aws:sns:us-east-1:825765388229:UserNotifications

Protocol
The type of endpoint to subscribe
Email

Endpoint
An email address that can receive notifications from Amazon SNS.
2022.snehal.patil@ves.ac.in

After your subscription is created, you must confirm it. Info



New Feature

Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more

Amazon SNS > Topics > UserNotifications > Subscription: 161e1e76-9161-4e6f-b827-7aa70711e879

Subscription: 161e1e76-9161-4e6f-b827-7aa70711e879

Details

ARN: arn:aws:sns:us-east-1:825765388229:UserNotifications:161e1e76-9161-4e6f-b827-7aa70711e879

Endpoint: 2022.snehal.patil@ves.ac.in

Topic: UserNotifications

Subscription Principal: arn:aws:iam::825765388229:root

Status: Confirmed

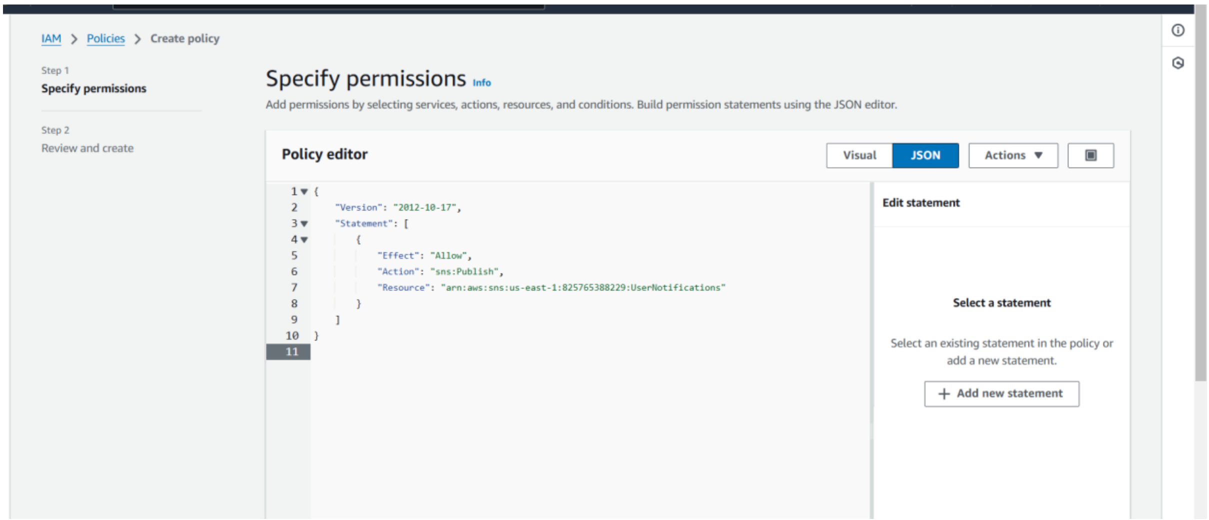
Protocol: EMAIL

## Update IAM Role for Lambda Functions

1. **Open IAM Management Console**:
   - Go to the IAM Console.
   - Select **Roles** in the left-hand menu.
   - Find and select the role associated with your Lambda function.
2. **Attach SNS Publish Policy**:
   - In the Permissions tab, click **Add permissions** -> **Attach policies**.
   - Click **Create policy**.
   - Switch to the **JSON** tab and paste the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
            "Action": "sns:Publish",
            "Resource": "arn:aws:sns:us-east-1:YOUR_ACCOUNT_ID:UserNotifications"
        }
    ]
}
```

- Click **Review policy**, name it (e.g., SNSPublishPolicy), and create the policy.
- Attach this policy to your Lambda function role.

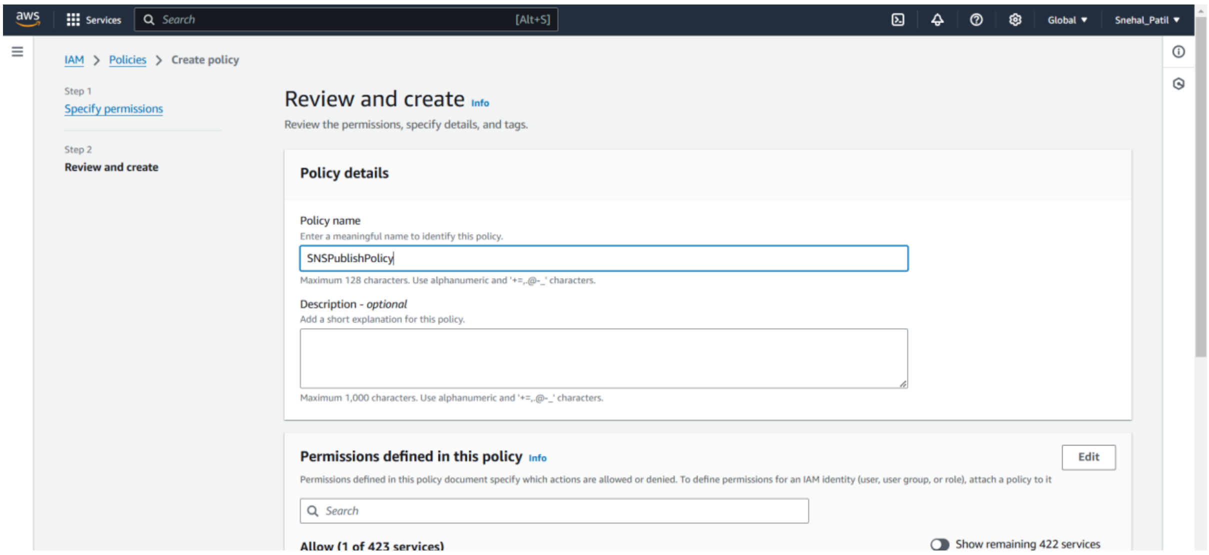


Specify permissions Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statement using the JSON editor.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sns:Publish",
            "Resource": "arn:aws:sns:us-east-1:825765388229:UserNotifications"
        }
    ]
}
```

Select an existing statement in the policy or add a new statement.



Review and create Info

Review the permissions, specify details, and tags.

Policy details

Policy name

Enter a meaningful name to identify this policy.

SNSPublishPolicy

Maximum 128 characters. Use alphanumeric and '+=,.@-_' characters.

Description - *optional*

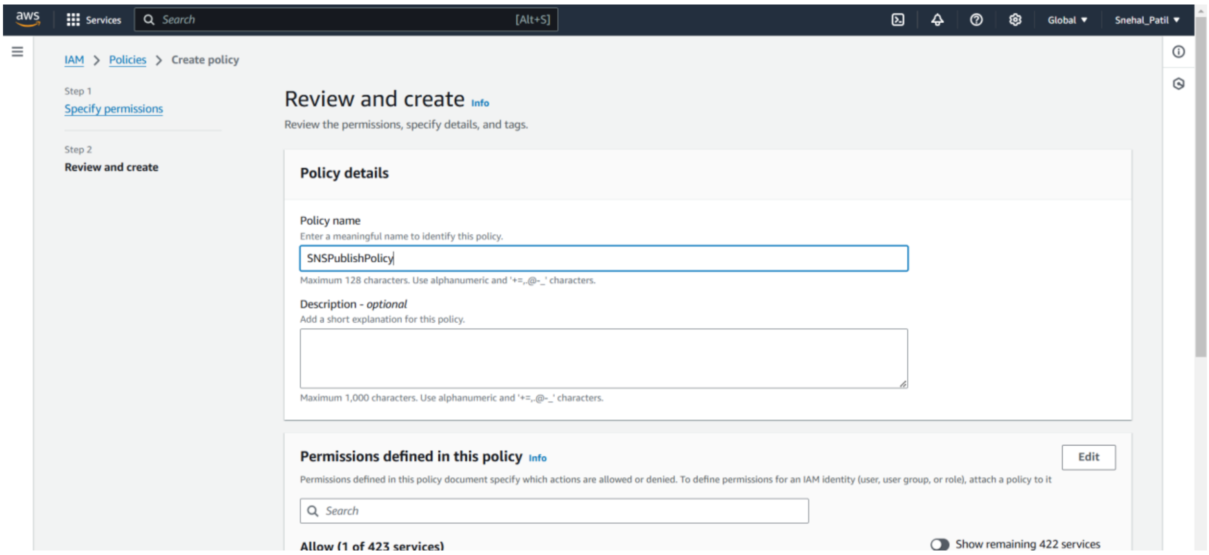
Add a short explanation for this policy.

Maximum 1,000 characters. Use alphanumeric and '+=,.@-_' characters.

Permissions defined in this policy Info

Permissions defined in this policy document specify which actions are allowed or denied. To define permissions for an IAM identity (user, user group, or role), attach a policy to it

Allow (1 of 423 services)



IAM > Policies > Create policy

Step 1
Specify permissions

Step 2
Review and create

## Review and create

Review the permissions, specify details, and tags.

**Policy details**

Policy name
Enter a meaningful name to identify this policy.
SNSPublishPolicy
Maximum 128 characters. Use alphanumeric and '+=,.@-_' characters.

Description - *optional*
Add a short explanation for this policy.

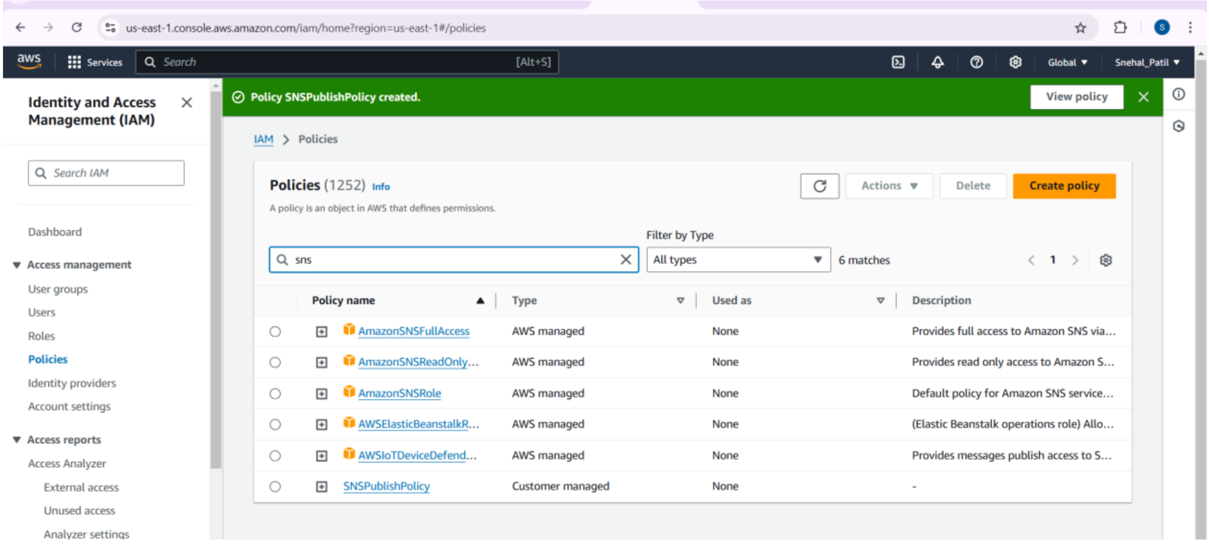
Maximum 1,000 characters. Use alphanumeric and '+=,.@-_' characters.

**Permissions defined in this policy** Info

Permissions defined in this policy document specify which actions are allowed or denied. To define permissions for an IAM identity (user, user group, or role), attach a policy to it

Allow (1 of 423 services)

Show remaining 422 services



us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/policies

Policy SNSPublishPolicy created.

IAM > Policies

**Policies** (1252) Info

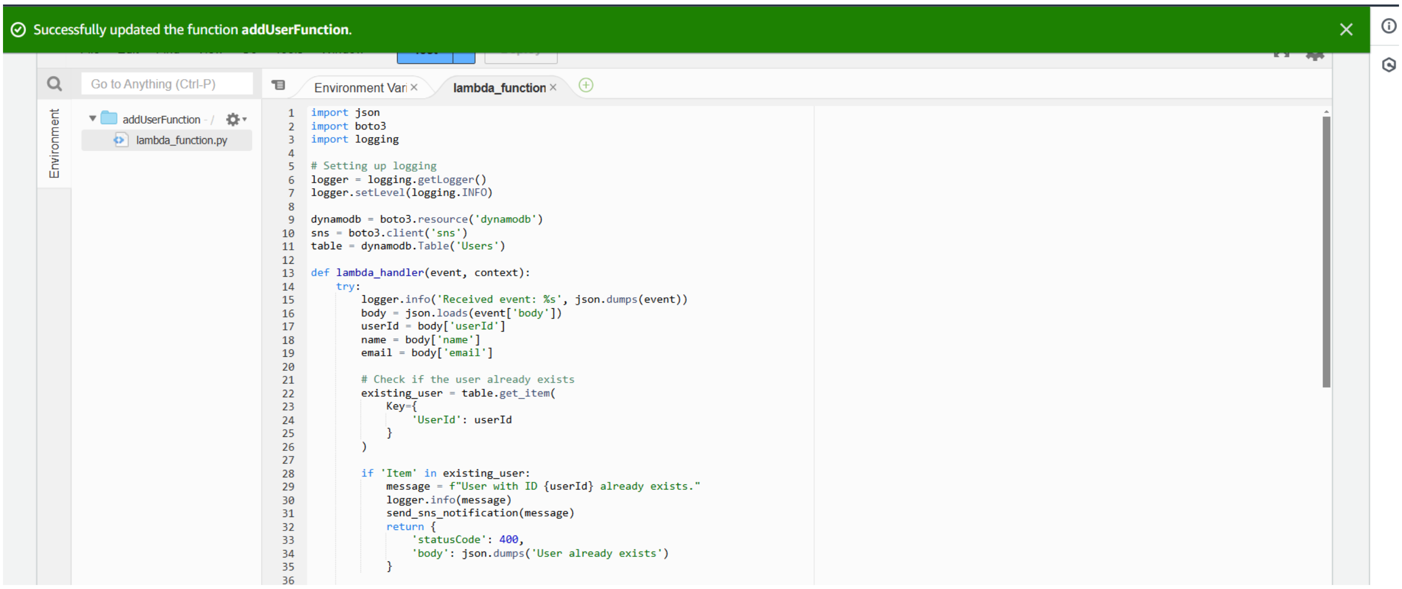
A policy is an object in AWS that defines permissions.

Filter by Type: All types

sns — 6 matches

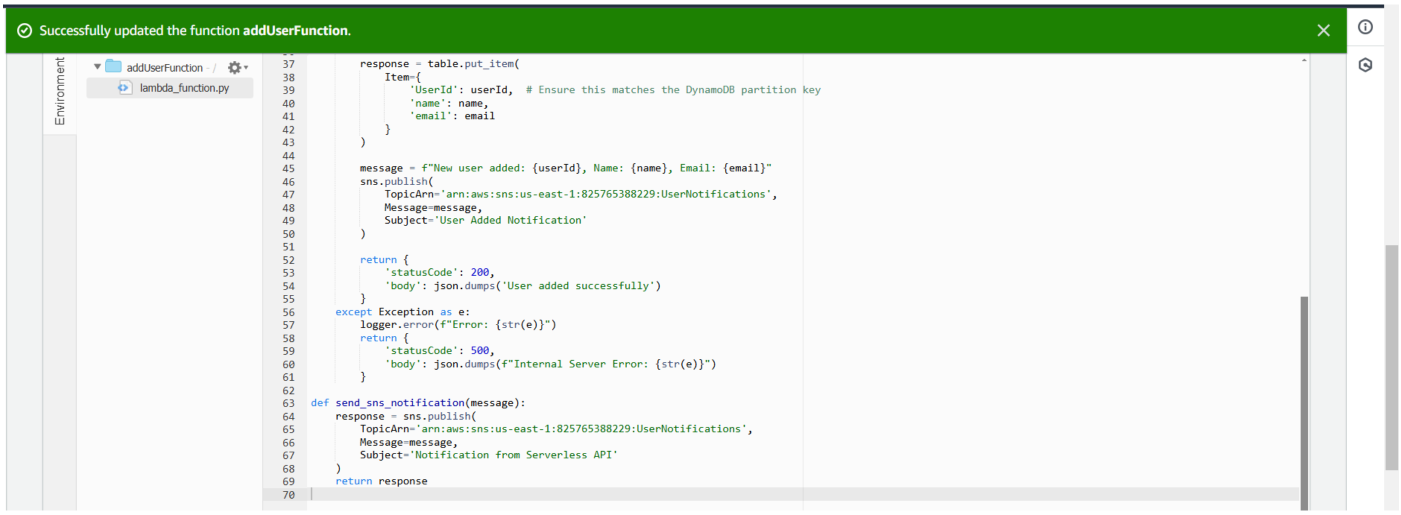
| | Policy name | Type | Used as | Description |
|---|---|---|---|---|
| ○ | AmazonSNSFullAccess | AWS managed | None | Provides full access to Amazon SNS via... |
| ○ | AmazonSNSReadOnly... | AWS managed | None | Provides read only access to Amazon S... |
| ○ | AmazonSNSRole | AWS managed | None | Default policy for Amazon SNS service... |
| ○ | AWSElasticBeanstalkR... | AWS managed | None | (Elastic Beanstalk operations role) Allo... |
| ○ | AWSIoTDeviceDefend... | AWS managed | None | Provides messages publish access to S... |
| ○ | SNSPublishPolicy | Customer managed | None | - |

## Updated  script in lambda function:



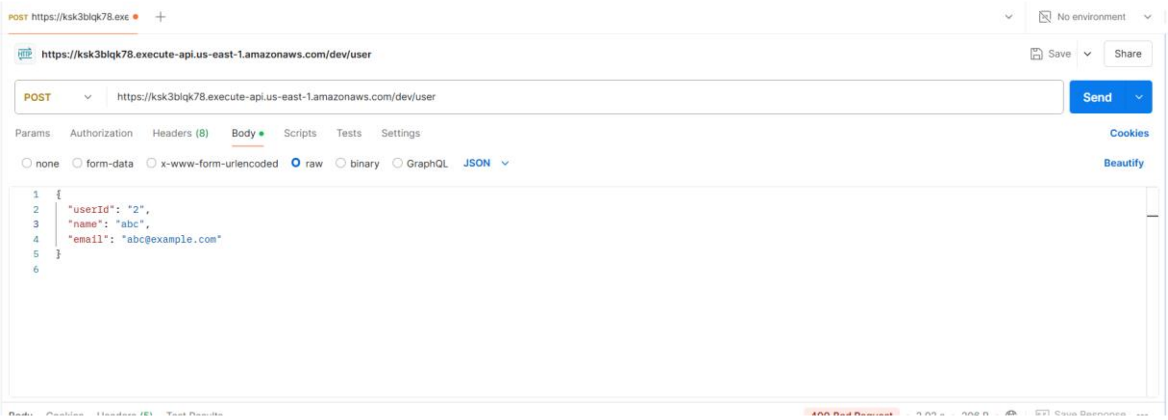
Successfully updated the function **addUserFunction**.

```python
import json
import boto3
import logging

# Setting up logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

dynamodb = boto3.resource('dynamodb')
sns = boto3.client('sns')
table = dynamodb.Table('Users')

def lambda_handler(event, context):
    try:
        logger.info('Received event: %s', json.dumps(event))
        body = json.loads(event['body'])
        userId = body['userId']
        name = body['name']
        email = body['email']

        # Check if the user already exists
        existing_user = table.get_item(
            Key={
                'UserId': userId
            }
        )

        if 'Item' in existing_user:
            message = f"User with ID {userId} already exists."
            logger.info(message)
            send_sns_notification(message)
            return {
                'statusCode': 400,
                'body': json.dumps('User already exists')
            }

```



```
        response = table.put_item(
            Item={
                'UserId': userId,  # Ensure this matches the DynamoDB partition key
                'name': name,
                'email': email
            }
        )

        message = f"New user added: {userId}, Name: {name}, Email: {email}"
        sns.publish(
            TopicArn='arn:aws:sns:us-east-1:825765388229:UserNotifications',
            Message=message,
            Subject='User Added Notification'
        )

        return {
            'statusCode': 200,
            'body': json.dumps('User added successfully')
        }
    except Exception as e:
        logger.error(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps(f"Internal Server Error: {str(e)}")
        }

def send_sns_notification(message):
    response = sns.publish(
        TopicArn='arn:aws:sns:us-east-1:825765388229:UserNotifications',
        Message=message,
        Subject='Notification from Serverless API'
    )
    return response
```

## User added successfully:



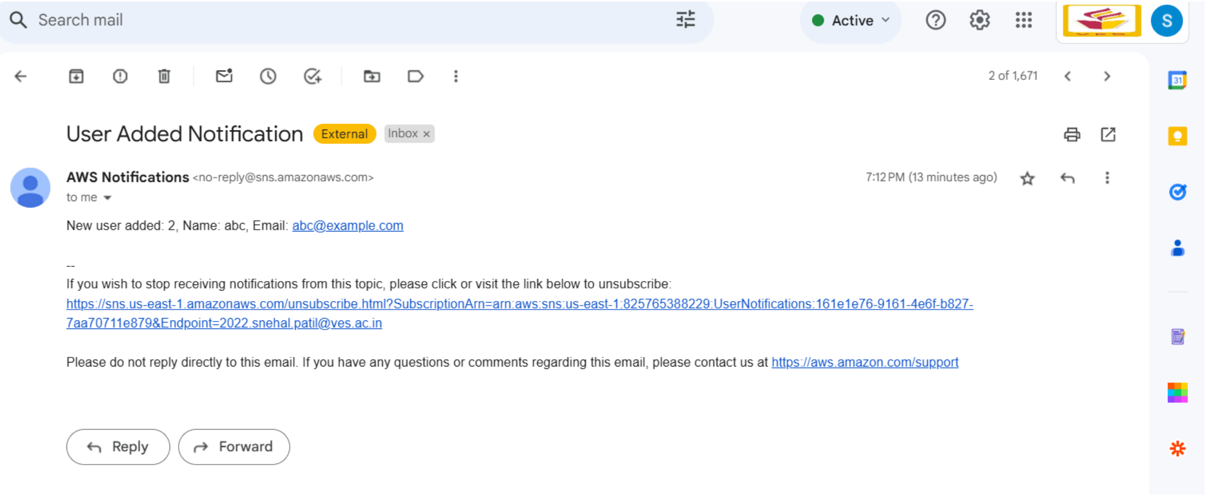
https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

POST https://ksk3blqk78.execute-api.us-east-1.amazonaws.com/dev/user

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
  "userId": "2",
  "name": "abc",
  "email": "abc@example.com"
}
```

## Email notification of user added:



User Added Notification External Inbox ×

AWS Notifications <no-reply@sns.amazonaws.com>
to me

7:12 PM (13 minutes ago)

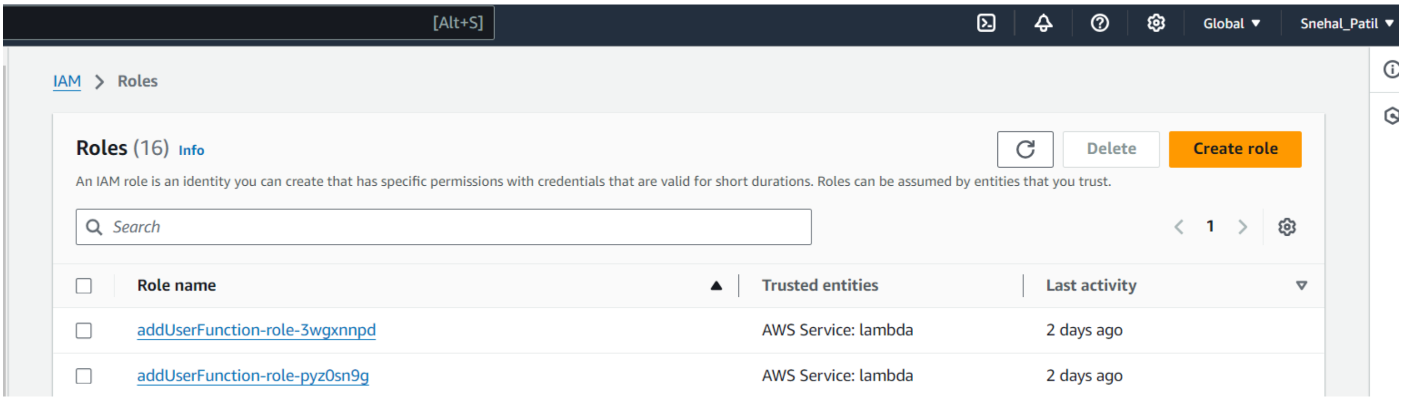
New user added: 2, Name: abc, Email: abc@example.com

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1:825765388229:UserNotifications:161e1e76-9161-4e6f-b827-7aa70711e879&Endpoint=2022.snehal.patil@ves.ac.in

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

## Guidelines

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

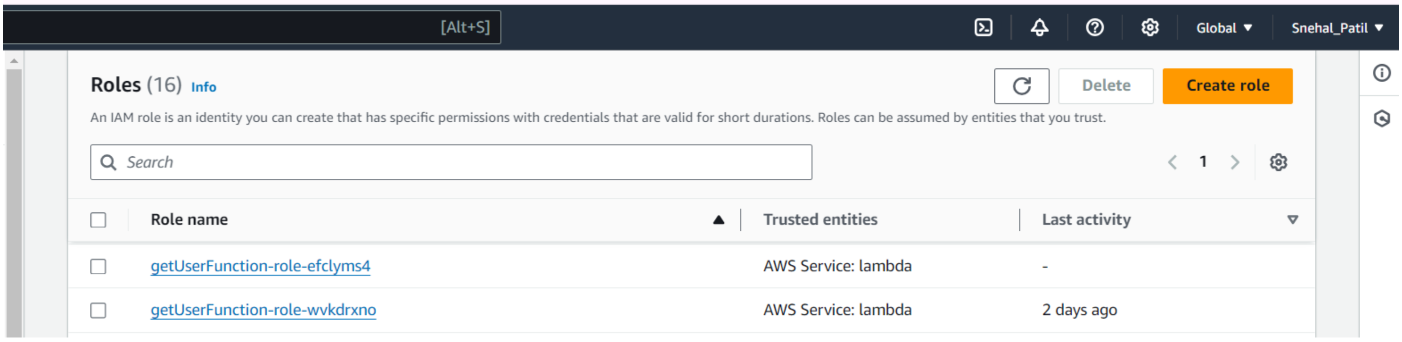


IAM > Roles

**Roles** (16) Info

An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

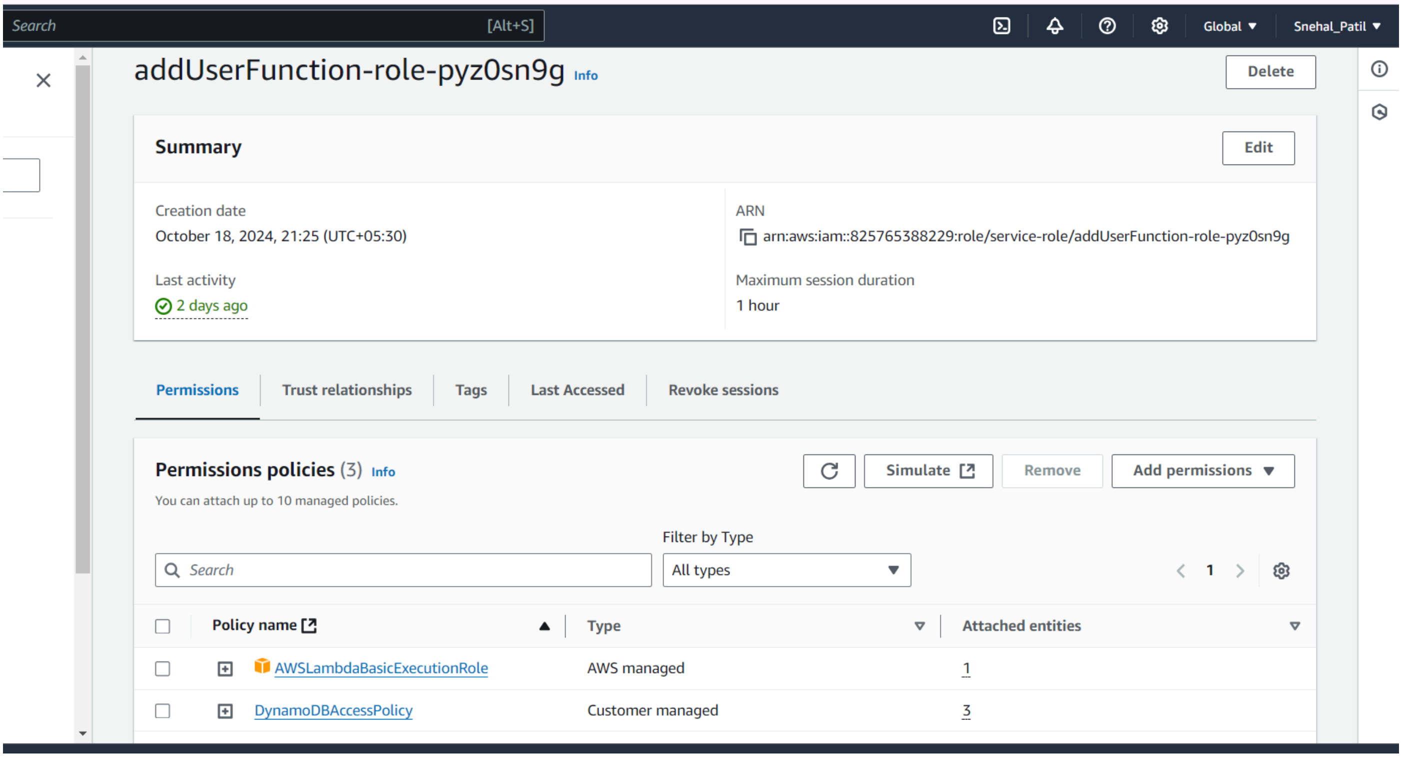
| □ | Role name | Trusted entities | Last activity |
|---|---|---|---|
| □ | addUserFunction-role-3wgxnnpd | AWS Service: lambda | 2 days ago |
| □ | addUserFunction-role-pyz0sn9g | AWS Service: lambda | 2 days ago |



**Roles** (16) Info

An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

| □ | Role name | Trusted entities | Last activity |
|---|---|---|---|
| □ | getUserFunction-role-efclyms4 | AWS Service: lambda | - |
| □ | getUserFunction-role-wvkdrxno | AWS Service: lambda | 2 days ago |



**addUserFunction-role-pyz0sn9g** Info

**Summary**

Creation date
October 18, 2024, 21:25 (UTC+05:30)

Last activity
2 days ago

ARN
arn:aws:iam::825765388229:role/service-role/addUserFunction-role-pyz0sn9g

Maximum session duration
1 hour

Permissions | Trust relationships | Tags | Last Accessed | Revoke sessions

**Permissions policies** (3) Info

You can attach up to 10 managed policies.

Filter by Type: All types

| □ | Policy name | Type | Attached entities |
|---|---|---|---|
| □ | AWSLambdaBasicExecutionRole | AWS managed | 1 |
| □ | DynamoDBAccessPolicy | Customer managed | 3 |

**Problems I Faced During execution :**

**Problem 1: Permissions Error**

- **Issue**: Encountered AccessDeniedException 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: 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 Gateway, 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.