

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 Gateway, 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 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**.

The screenshot shows the 'Create table' page in the AWS Management Console. The breadcrumb navigation is 'DynamoDB > Tables > Create table'. The page title is 'Create table'. Below the title, there is a 'Table details' section with an 'Info' icon. The text states: 'DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.' The 'Table name' field is labeled 'Table name' and 'This will be used to identify your table.' The value 'Users' is entered. Below the field, it says 'Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).' The 'Partition key' section is labeled 'Partition key' and '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.' The value 'UserId' is entered in the text box, and 'String' is selected in the dropdown menu. Below the field, it says '1 to 255 characters and case sensitive.' The 'Sort key - optional' section is labeled 'Sort key - optional' and '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.' The text box contains 'Enter the sort key name' and 'String' is selected in the dropdown menu. Below the field, it says '1 to 255 characters and case sensitive.'

The screenshot shows the 'Tables' page in the AWS Management Console. The breadcrumb navigation is 'DynamoDB > Tables'. The page title is 'Tables (1) Info'. There is a search bar with the text 'Find tables'. Below the search bar, there is a table with the following columns: Name, Status, Partition key, Sort key, Indexes, Deletion protection, Favorite, Read capacity mode, and Write capacity mode. The table has one row with the following values: Name: Users, Status: Active, Partition key: UserId (S), Sort key: -, Indexes: 0, Deletion protection: Off, Favorite: ☆, Read capacity mode: Provisioned (1), and Write capacity mode: Provisioned (1). The table is sorted by Name in ascending order. The page also has a 'Create table' button and a 'Delete' button.

	Name ▲	Status ▼	Partition key ▼	Sort key ▼	Indexes ▼	Deletion protection ▼	Favorite ▼	Read capacity mode ▼	Write capacity mode ▼
<input type="checkbox"/>	Users	Active	UserId (S)	-	0	Off	☆	Provisioned (1)	Provisioned (1)

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

aws Services Search [Alt+S]

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

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

aws Services Search [Alt+S]

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:

Services Search [Alt+S]

Code source Info

File Edit Find View Go Tools Window

Go to Anything (Ctrl-P)

Environment

addUserFunction - /

```

1 import json
2 import boto3
3
4 dynamodb = boto3.resource('dynamodb')
5 table = dynamodb.Table('Users')
6
7 def lambda_handler(event, context):
8     body = json.loads(event['body'])
9     userId = body['userId']
10    name = body['name']
11    email = body['email']
12
13    response = table.put_item(
14        Item={
15            'UserId': userId,
16            'name': name,
17            'email': email
18        }
19    )
20
21    return {
22        'statusCode': 200,
23        'body': json.dumps('User added successfully')
24    }
25
  
```

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.

The screenshot shows the 'Create function' page in the AWS Lambda console. The 'Author from scratch' option is selected. The 'Basic information' section shows the function name 'getUserFunction', the runtime 'Python 3.12', and the architecture 'x86_64'.

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**

Code for getUserFunction:

The screenshot shows the 'Code source' page in the AWS Lambda console. The code editor displays the Python code for the getUserFunction, which uses boto3 to interact with DynamoDB.

Code source [Info](#)

Upload from ▾

File Edit Find View Go Tools Window Test Deploy

Go to Anything (Ctrl-P)

Environment Var x lambda_function x

```
1 import json
2 import boto3
3
4 dynamodb = boto3.resource('dynamodb')
5 table = dynamodb.Table('Users')
6
7 def lambda_handler(event, context):
8     userId = event['pathParameters']['userId']
9
10    response = table.get_item(
11        Key={
12            'UserId': userId
13        }
14    )
15
16    if 'Item' in response:
17        return {
18            'statusCode': 200,
19            'body': json.dumps(response['Item'])
20        }
21    else:
22        return {
23            'statusCode': 404,
24            'body': json.dumps('User not found')
25        }
26
```

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).

The screenshot shows the AWS API Gateway console. The breadcrumb trail is 'API Gateway > APIs > Create API > Create'. The left sidebar shows the steps: Step 1: Create an API (active), Step 2 - optional: Configure routes, Step 3 - optional: Define stages, and Step 4: Review and Create. The main content area is titled 'Create an API' and contains a section 'Create and configure integrations'. This section explains that integrations are backend services the API will communicate with. Below this, there is a section 'Integrations (0) Info' with an 'Add integration' button. At the bottom, there is a section 'API name' with a text box containing 'UserAPI' and a 'Next' button.

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.

The screenshot shows the 'Create a route' page in the AWS API Gateway console. The title is 'Create a route'. Below it is a section 'Route and method Info'. This section contains a text box for 'Route name eg. /pets' and a description: '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.' Below this, there is a dropdown menu for the method, currently set to 'POST', and a text box for the path, currently set to '/user'. At the bottom right, there are 'Cancel' and 'Create' buttons.

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

Create an integration

Attach this integration to a route

Q 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

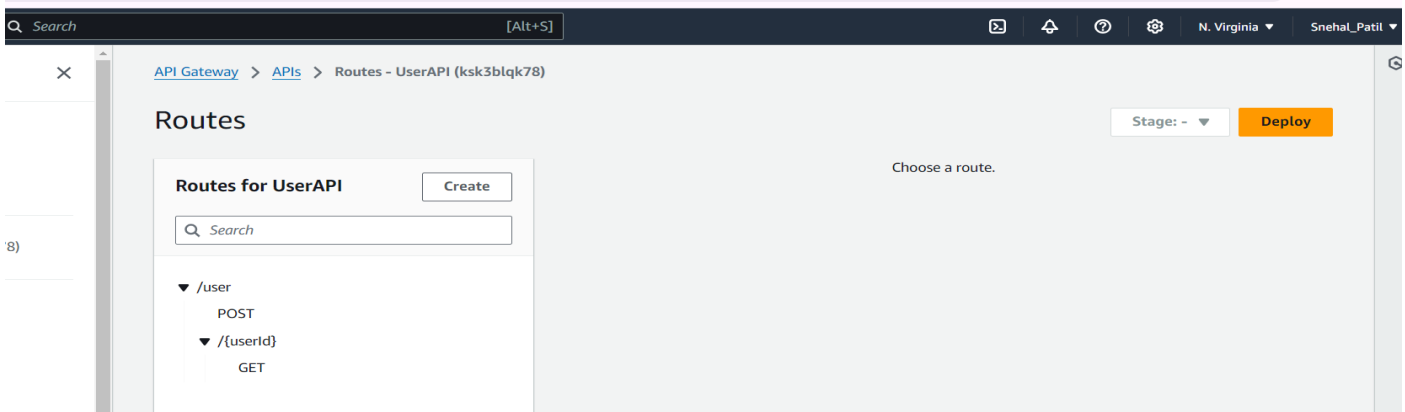
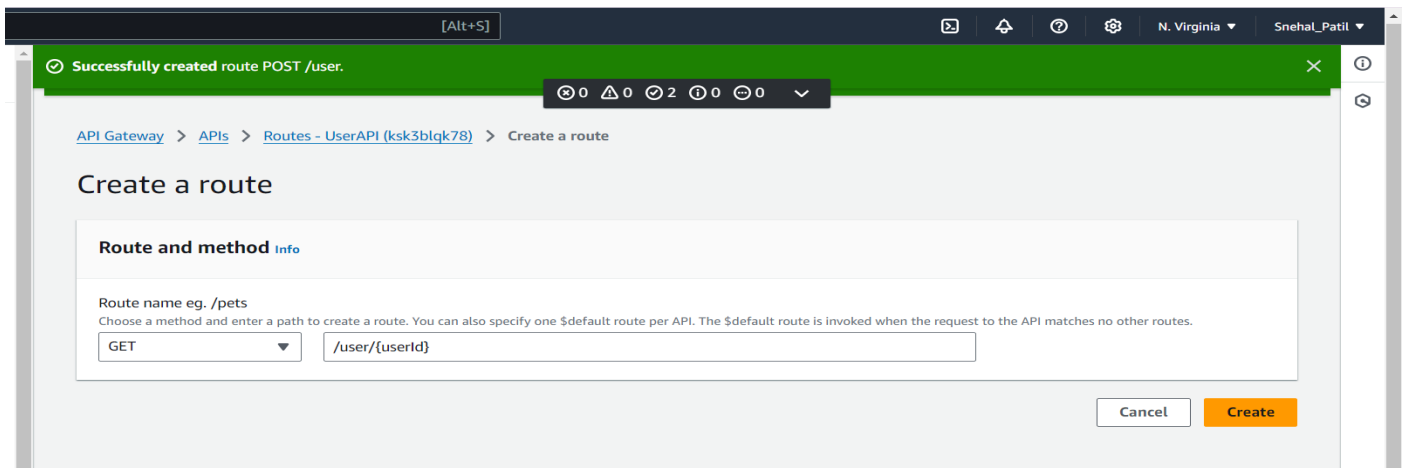
API Gateway requests permission to invoke your Lambda function by granting this API permission in your Lambda function's resource policy. If you don't want to modify the resource policy, you can provide an invoke role instead. API Gateway uses this role to invoke the Lambda function.

☒ Grant API Gateway permission to invoke your Lambda function

Cancel Create

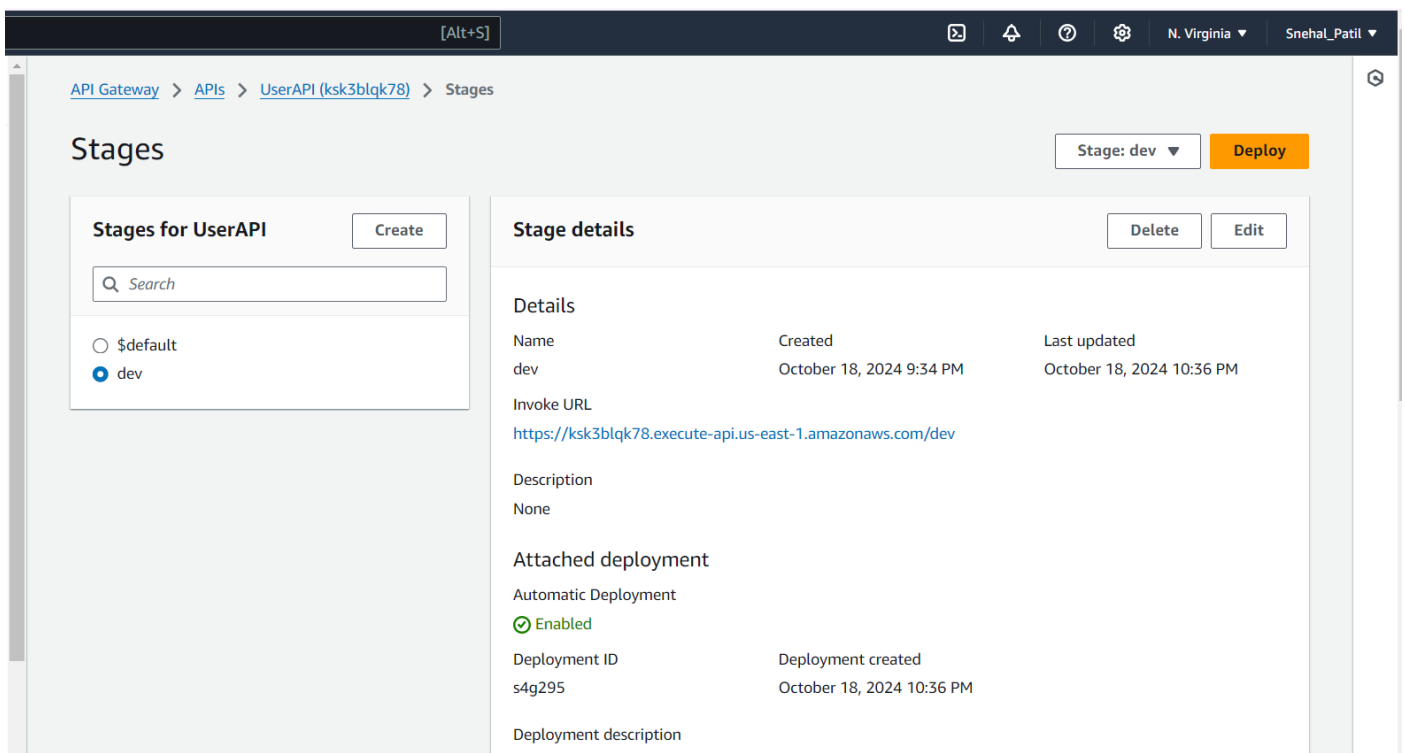
Creating GET Method for Retrieving User by ID

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**.

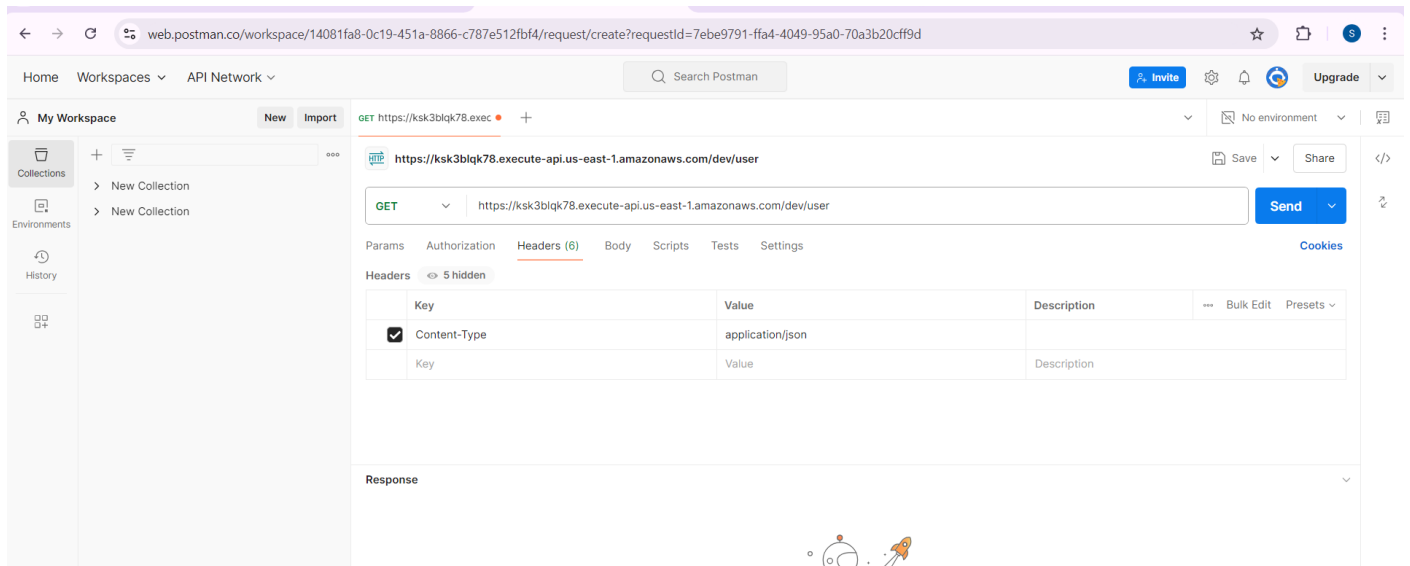


Deploying the API

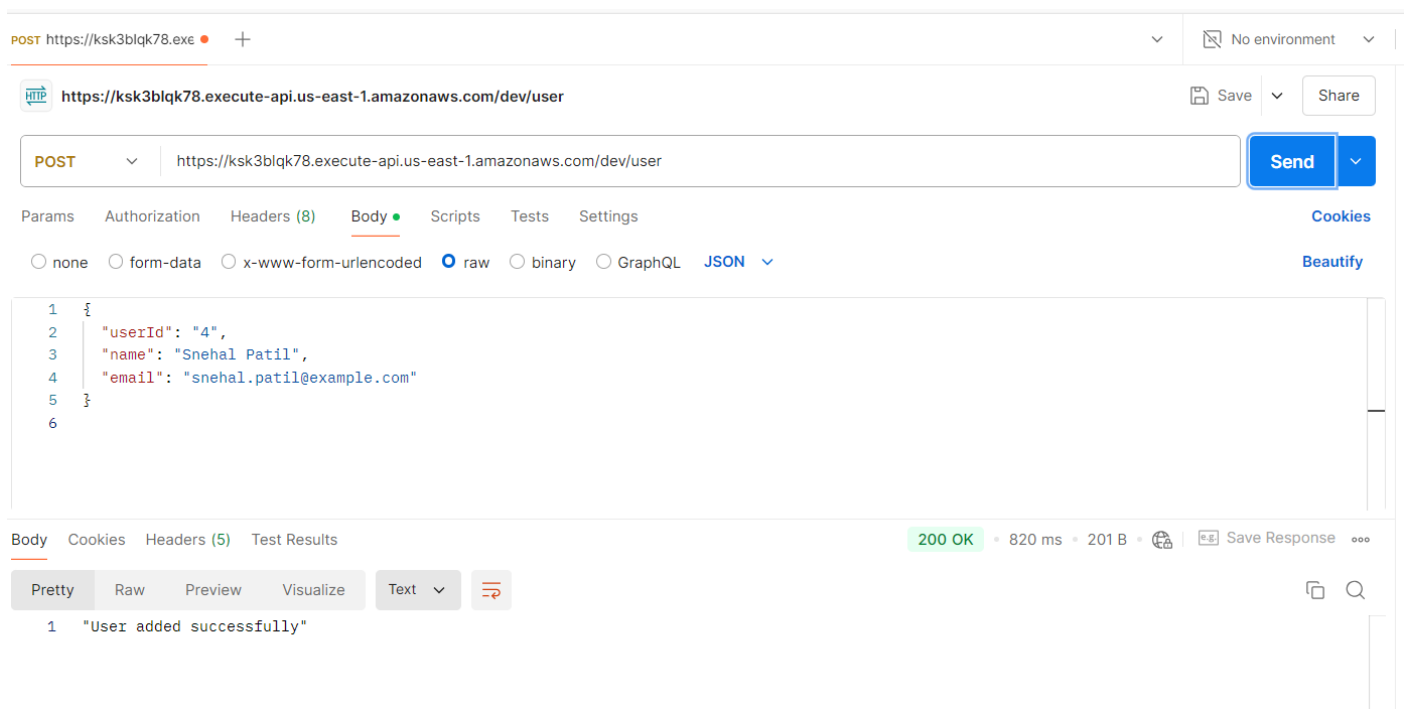
1. Click **Deploy API**.
2. Create a new deployment stage (e.g., dev).
3. You will receive the API endpoint URL once deployed.



Testing the api on postman:



Post request to add data to the dynamo table



Get request to fetch the user data from table

GET https://ksk3blqk78.exec +

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

Save Share

GET

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

Send

Params

Authorization

Headers (8)

Body

Scripts

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

1

{

2

"userId": "4",

3

"name": "Snehal Patil",

4

"email": "snehal.patil@example.com"

5

}

6

Body

Cookies

Headers (5)

Test Results

200 OK

770 ms

247 B

Save Response

Pretty

Raw

Preview

Visualize

Text

1

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

Trying to fetch the data not present in the table

GET https://ksk3blqk78.exec +

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

Save Share

GET

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

Send

Params

Authorization

Headers (8)

Body

Scripts

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

1

{

2

"userId": "4",

3

"name": "Snehal Patil",

4

"email": "snehal.patil@example.com"

5

}

6

Body

Cookies

Headers (5)

Test Results

404 Not Found

779 ms

194 B

Save Response

Pretty

Raw

Preview

Visualize

Text

1

"User not found"

Guidelines

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

[Alt+S]

GlobalSneha_Patil

IAM > Roles

Roles (16) Info

Refresh

Delete

Create role

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.

Search

< 1 > Settings

<input type="checkbox"/>	Role name	Trusted entities	Last activity
<input type="checkbox"/>	addUserFunction-role-3wgxnnpd	AWS Service: lambda	2 days ago
<input type="checkbox"/>	addUserFunction-role-pyz0sn9g	AWS Service: lambda	2 days ago

[Alt+S]

GlobalSneha_Patil

Roles (16) Info

Refresh

Delete

Create role

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.

Search

< 1 > Settings

<input type="checkbox"/>	Role name	Trusted entities	Last activity
<input type="checkbox"/>	getUserFunction-role-efclyms4	AWS Service: lambda	-
<input type="checkbox"/>	getUserFunction-role-wvkdrnxo	AWS Service: lambda	2 days ago

Search

[Alt+S]

GlobalSneha_Patil

addUserFunction-role-pyz0sn9g Info

Delete

Summary

Edit

Creation date

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

ARN

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

Last activity

2 days ago

Maximum session duration

1 hour

Permissions

Trust relationships

Tags

Last Accessed

Revoke sessions

Permissions policies (3) Info

Refresh

Simulate

Remove

Add permissions

You can attach up to 10 managed policies.

Search

Filter by Type

All types

< 1 > Settings

<input type="checkbox"/>	Policy name	Type	Attached entities
<input type="checkbox"/>	AWSLambdaBasicExecutionRole	AWS managed	1
<input type="checkbox"/>	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.