Name: Alok Yadav Div : D15C Roll No.:59

Building a Serverless REST API

1. Introduction

Case Study Overview:

This case study focuses on creating a serverless REST API using the Serverless Framework, AWS Lambda, and API Gateway to manage user data in a DynamoDB table. The API allows for adding new users and retrieving user details.

Key Feature and Application:

The key feature of this application is its ability to seamlessly perform CRUD operations (Create, Read, Update, and Delete) on user data, leveraging a serverless architecture. This architecture eliminates the need for traditional server management, reducing the operational overhead and costs associated with maintaining dedicated infrastructure. By using AWS Lambda to execute backend logic and Amazon DynamoDB for fast and scalable data storage, the application can dynamically handle user data in real-time. Furthermore, the integration with AWS API Gateway allows secure and efficient communication between the frontend and backend, ensuring smooth data flow for operations such as user profile management, authentication, and session tracking. This serverless approach is particularly well-suited for applications that require flexibility and scalability, enabling automatic handling of varying workloads without manual intervention, making it ideal for modern applications with fluctuating traffic or growing user bases.

2. Setup of serverless REST API

Step 1: Initial Setup

1. Install Serverless Framework:

```
> npm install -g serverless
```

2. Create a New Serverless Project:

```
> serverless create --template aws-nodejs --path restapi-serverless
> cd restapi-serverless
> npm init -y
> npm install aws-sdk
```

Note: Installation of aws-sdk is necessary to interact with aws services
After installing serverless configure it with AWS credentials for deployment using command **aws configure**

3. Configure `serverless.yml`:
   Open the **serverless.yml** file and configure it to set up the DynamoDB table and Lambda functions:

```
service: serverless-rest-api

provider:
  name: aws
  runtime: nodejs18.x
  region: us-east-1

functions:
  addUser:
    handler: handler.addUser
    events:
      - http:
          path: users
          method: post
          cors: true

  getUser:
    handler: handler.getUser
    events:
      - http:
          path: users/{id}
          method: get
          cors: true

resources:
  Resources:
    UsersTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: Users
        AttributeDefinitions:
          - AttributeName: UserID
            AttributeType: S
        KeySchema:
          - AttributeName: UserID
            KeyType: HASH
        BillingMode: PAY_PER_REQUEST
```

Step 2: Write the Lambda Function to add and retrieve user information

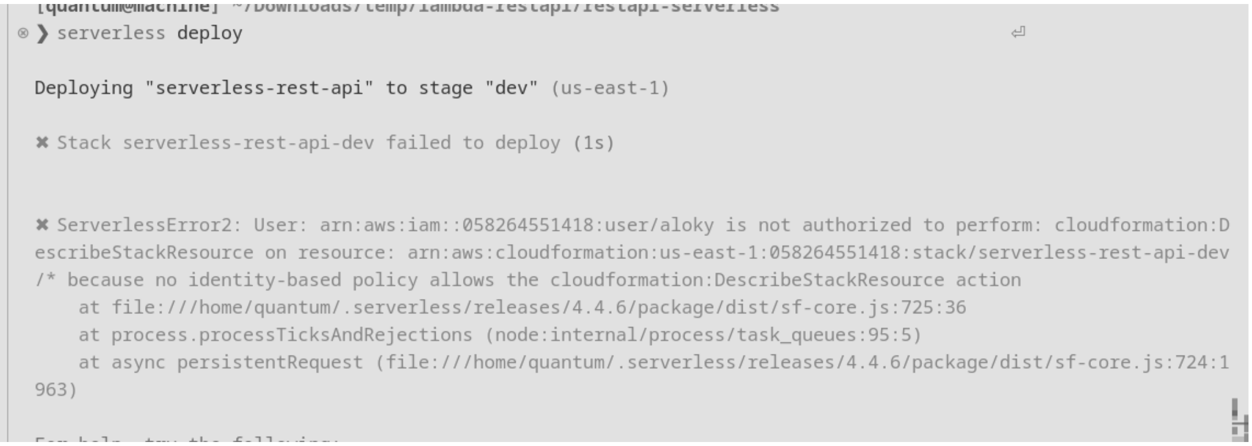
1. Implement the Lambda Function:
   Open the **handler.js** file and add the following code to handle adding a user to DynamoDB:

```
const AWS = require('aws-sdk');
const dynamo = new AWS.DynamoDB.DocumentClient();

module.exports.addUser = async (event) => {
  const { UserID, Name, Email } = JSON.parse(event.body);

  const params = {
    TableName: 'Users',
    Item: { UserID, Name, Email },
  };

  try {
    await dynamo.put(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify({ message: 'User added successfully!' }),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ message: 'Error adding user', error }),
    };
  }
};

module.exports.getUser = async (event) => {
  const { id } = event.pathParameters;

  const params = {
    TableName: 'Users',
    Key: { UserID: id },
  };

  try {
    const data = await dynamo.get(params).promise();
    if (data.Item) {
      return {
        statusCode: 200,
        body: JSON.stringify(data.Item),
      };
    } else {
      return {
        statusCode: 404,
        body: JSON.stringify({ message: 'User not found' }),
      };
    }
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ message: 'Error retrieving user', error }),
    };
  }
};
```

Step 3: Deploy the API

Run the following command to deploy your application to AWS:

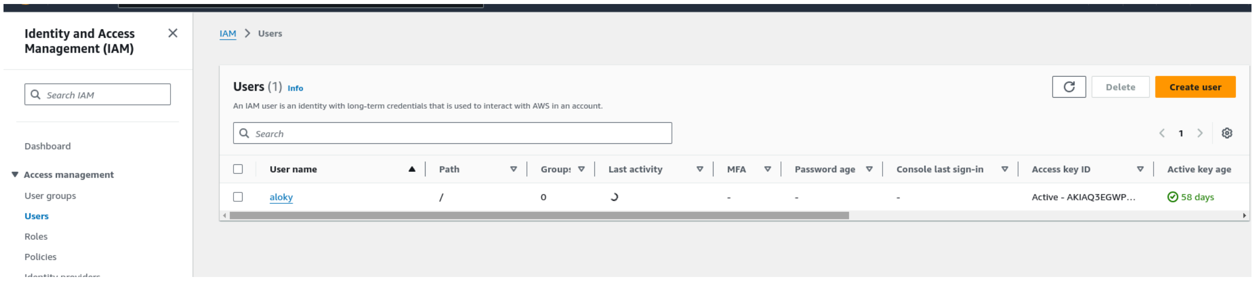
```
> serverless deploy
```



```
[quantum@machine] ~/Downloads/temp/lambda-restapi/restapi-serverless
⊗ ❯ serverless deploy

Deploying "serverless-rest-api" to stage "dev" (us-east-1)

✖ Stack serverless-rest-api-dev failed to deploy (1s)


✖ ServerlessError2: User: arn:aws:iam::058264551418:user/aloky is not authorized to perform: cloudformation:D
escribeStackResource on resource: arn:aws:cloudformation:us-east-1:058264551418:stack/serverless-rest-api-dev
/* because no identity-based policy allows the cloudformation:DescribeStackResource action
    at file:///home/quantum/.serverless/releases/4.4.6/package/dist/sf-core.js:725:36
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async persistentRequest (file:///home/quantum/.serverless/releases/4.4.6/package/dist/sf-core.js:724:1
963)

For help, try the following:
```

Here we got an error because the current user does not have access to modify the cloudformation stack.

To resolve this, Go to **IAM dashboard** and select users followed by user whose permission has to be changed

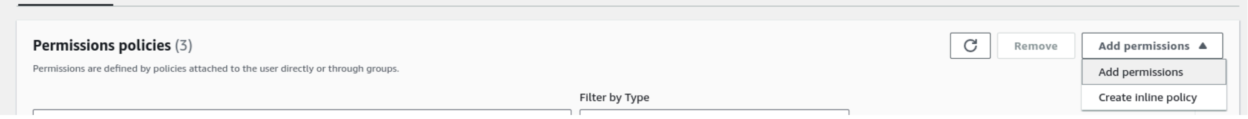


Users (1) Info

An IAM user is an identity with long-term credentials that is used to interact with AWS in an account.

□	User name	Path	Group:	Last activity	MFA	Password age	Console last sign-in	Access key ID	Active key age
□	aloky	/	0		-	-	-	Active - AKIAQ3EGWP...	✓ 58 days

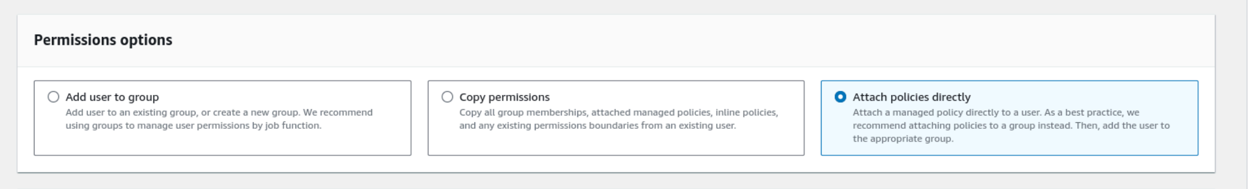
Under **Permissions policies,** click on **Add permissions**



Permissions policies (3)

Permissions are defined by policies attached to the user directly or through groups.

Then select **Attach policies directly**



Permissions options

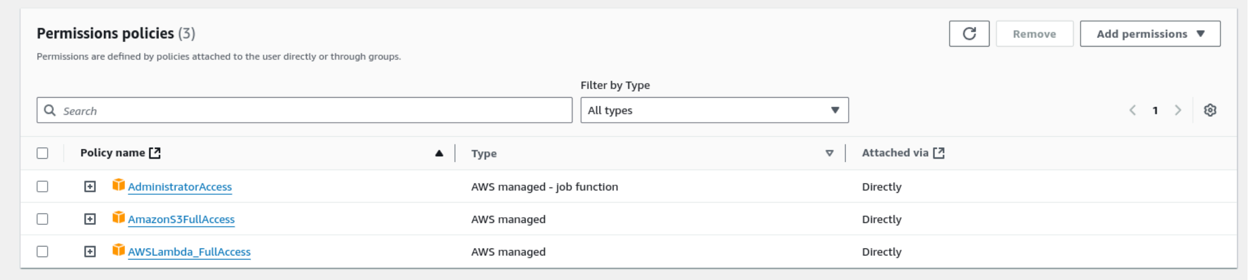
○ Add user to group
Add user to an existing group, or create a new group. We recommend using groups to manage user permissions by job function.

○ Copy permissions
Copy all group memberships, attached managed policies, inline policies, and any existing permissions boundaries from an existing user.

● Attach policies directly
Attach a managed policy directly to a user. As a best practice, we recommend attaching policies to a group instead. Then, add the user to the appropriate group.

Then search for **AdministratorAccess** and apply that policy

Also ensure that the user has permissions to other services such as AWS Lambda

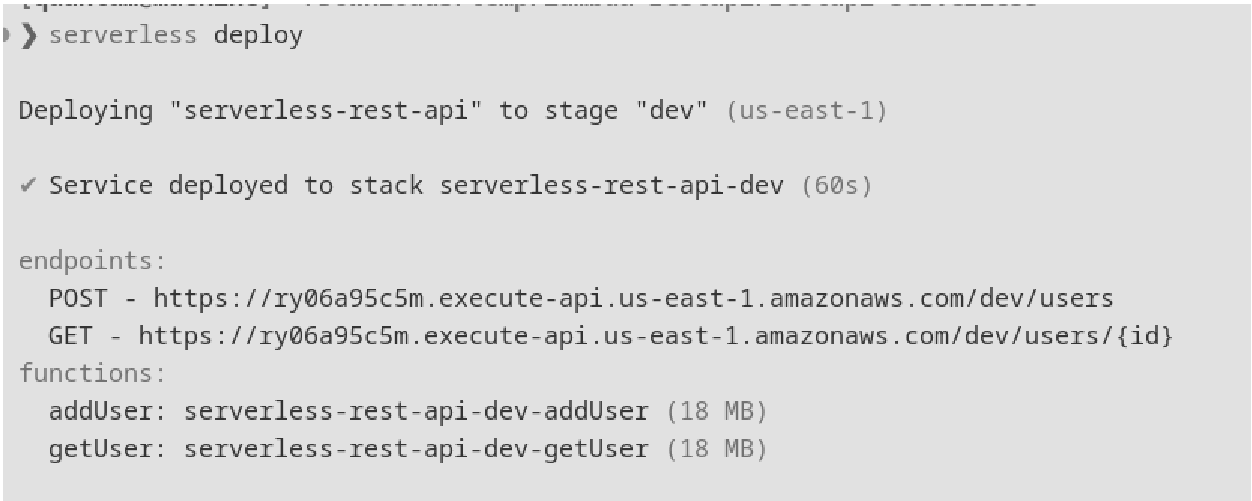


Permissions policies (3)

Permissions are defined by policies attached to the user directly or through groups.

□	Policy name	Type	Attached via
□	AdministratorAccess	AWS managed - job function	Directly
□	AmazonS3FullAccess	AWS managed	Directly
□	AWSLambda_FullAccess	AWS managed	Directly

After adding all necessary permissions, run command **serverless deploy** to deploy again



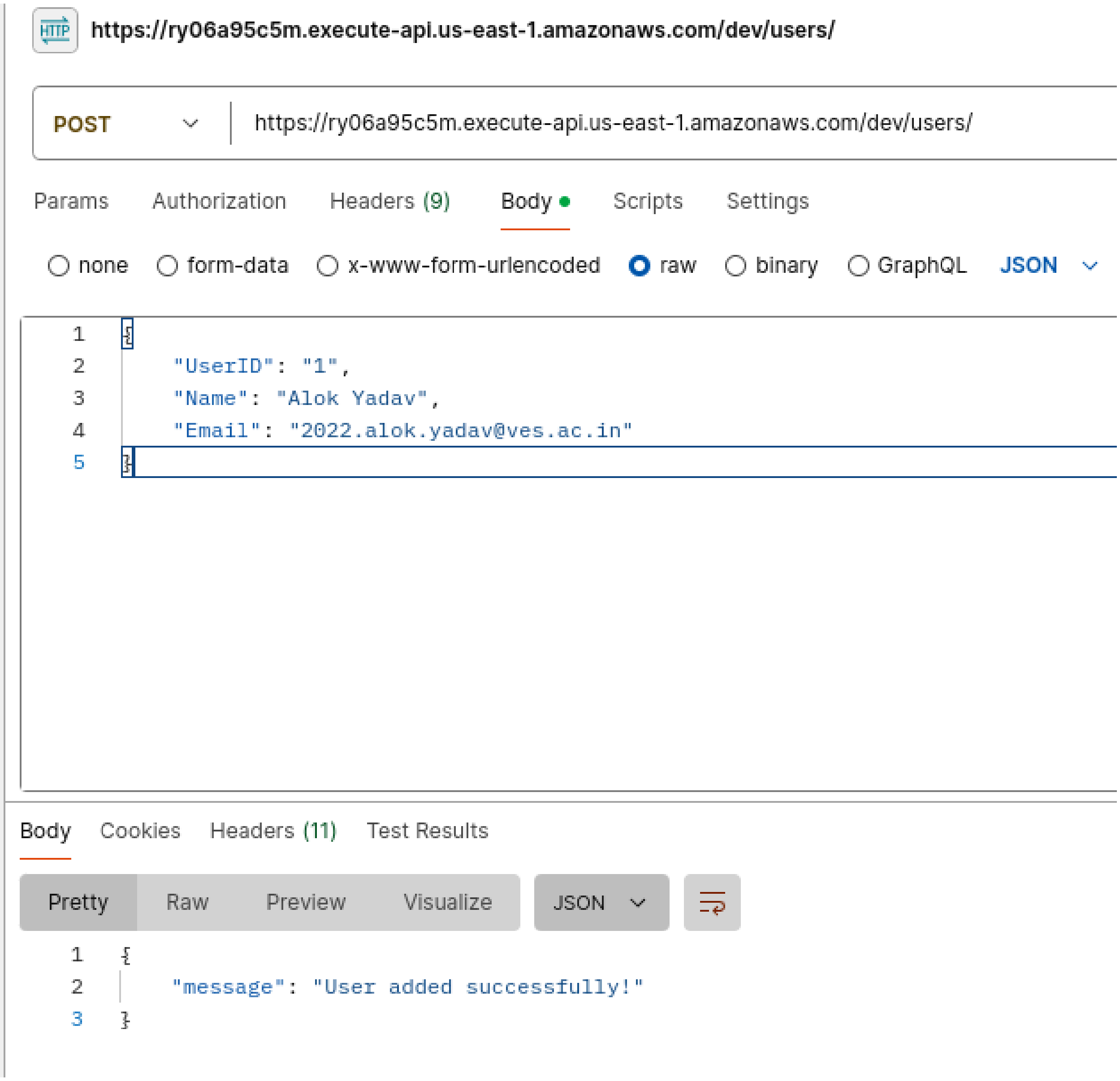
```
❯ serverless deploy

Deploying "serverless-rest-api" to stage "dev" (us-east-1)

✔ Service deployed to stack serverless-rest-api-dev (60s)

endpoints:
  POST - https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users
  GET - https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users/{id}
functions:
  addUser: serverless-rest-api-dev-addUser (18 MB)
  getUser: serverless-rest-api-dev-getUser (18 MB)
```

As we can see, our serverless REST API is successfully deployed and we get two API endpoints, one for adding users and other for retrieving user details.

Step 4: Testing the API

We can test our API using **Postman** or **CURL**

1. Testing API to add user into Amazon DynamoDB
   Open postman, in the URL bar input https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users with method as **POST** followed by user data in body. Click on send to send request



POST https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users/

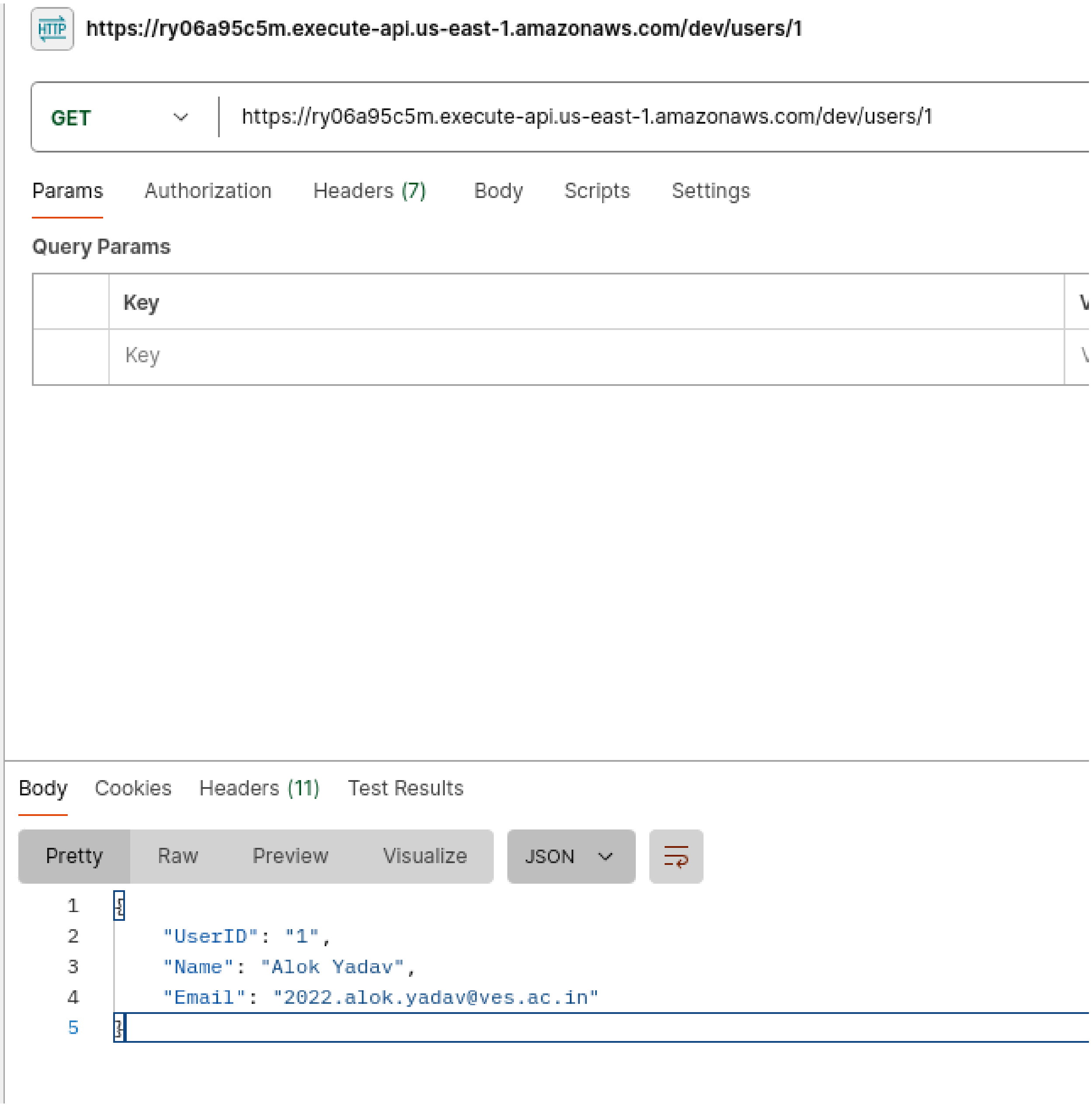
○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
    "UserID": "1",
    "Name": "Alok Yadav",
    "Email": "2022.alok.yadav@ves.ac.in"
}
```

```
{
    "message": "User added successfully!"
}
```

The new user is successfully added to the database

2. Testing API to retrieve added users
   Open postman, in the URL bar input https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users/1 with method type as **GET** then click on send



https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users/1

GET https://ry06a95c5m.execute-api.us-east-1.amazonaws.com/dev/users/1

```
{
    "UserID": "1",
    "Name": "Alok Yadav",
    "Email": "2022.alok.yadav@ves.ac.in"
}
```

The API was able to fetch added user from database

Name: Alok Yadav Div : D15C Roll No.:59

Third-Year Project Integration

Introduction :

Our project, **MentorLink**, is a platform where mentees and mentors can connect, share guidance, and gain valuable insights. Given the extensive CRUD operations required for managing user profiles, project data, authentication, and session management, we need a database that is fast and reliable. Amazon DynamoDB meets these requirements perfectly. Moreover, by integrating it with AWS Lambda functions and API Gateway, we can create serverless RESTful APIs, simplifying the process of handling all CRUD operations seamlessly

Steps to deploy a backend server as an AWS Lambda function integrated with AWS API Gateway :

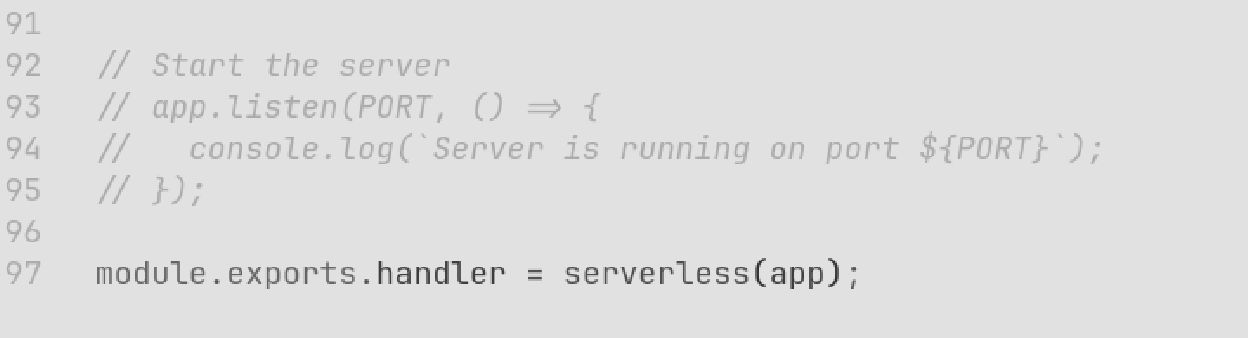
Prerequisites :
Serverless installed and configured with aws credentials

1. Set Up Serverless in Your Project using
   > **serverless**

2. Install Additional Packages
   To make the Express app compatible with AWS Lambda, we need the following package:
   > **npm install serverless-http**
   To test the serverless application locally before deploying on AWS we can install **serverless-offline** package
   > **npm install serverless-offline --save-dev**

3. Modify the main entry point file of your Express Application
   Replace the `app.listen()` line in your `server.js` or `main.js/index.js` with the following code:

```
const serverless = require('serverless-http');
 module.exports.handler = serverless(app);
```



```
// Start the server
// app.listen(PORT, () => {
//   console.log(`Server is running on port ${PORT}`);
// });

module.exports.handler = serverless(app);
```

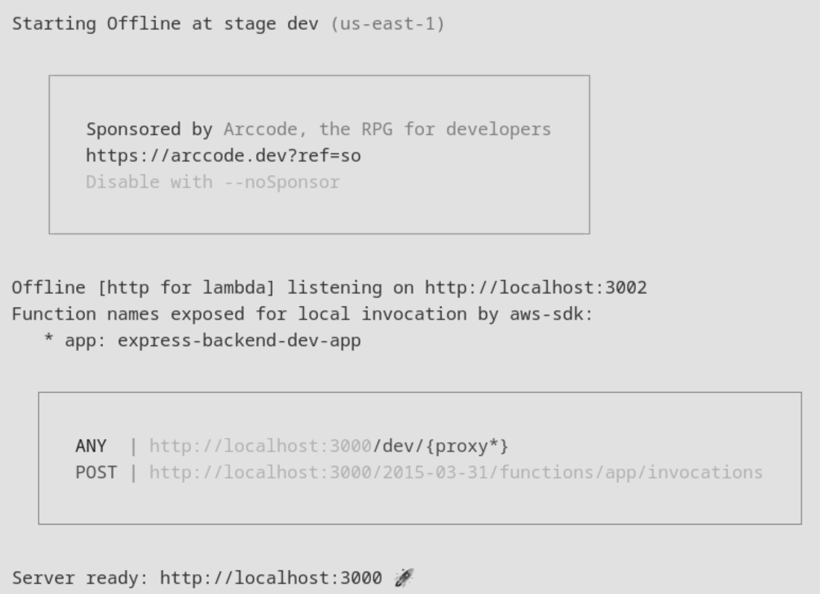
4. Create `serverless.yml` File
   Create a `serverless.yml` configuration file at the root of your project:

```
service: express-backend

provider:
  name: aws
  runtime: nodejs18.x
  stage: dev
  region: us-east-1
  environment:
    DB_USER : <user>
    DB_PASSWORD : <db_password>
    DB_HOST : <db_host_url>
    DB_NAME : mentorlink
    JWT_SECRET : <JWT_SECRET>
    SESSION_SECRET : SUPER_SECRET


functions:
  app:
    handler: server.handler
    events:
      - http:
          path: /{proxy+}
          method: any
          cors: true

plugins:
  - serverless-offline
```

```
package:
  exclude:
    - uploads/**
    - .git/**

custom:
  serverless-offline:
    useChildProcesses: true

# IAM permissions for AWS Lambda
iamRoleStatements:
  - Effect: Allow
    Action:
      - s3:PutObject
      - s3:GetObject
    Resource: "*"
```

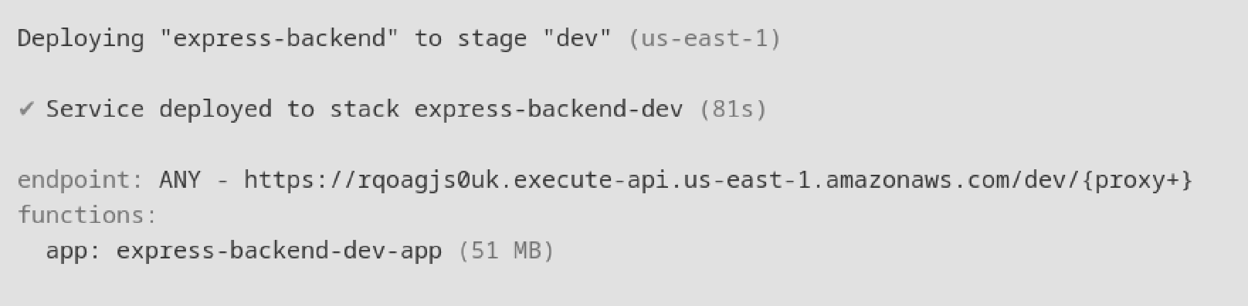
Note: Ensure you mention all the required details to run your project including environment variables

5. Test our serverless RESTfull application locally
   We can test our application locally before deploying using below command:
   > **serverless offline**



```
Starting Offline at stage dev (us-east-1)

    Sponsored by Arccode, the RPG for developers
    https://arccode.dev?ref=so
    Disable with --noSponsor

Offline [http for lambda] listening on http://localhost:3002
Function names exposed for local invocation by aws-sdk:
   * app: express-backend-dev-app

   ANY  | http://localhost:3000/dev/{proxy*}
   POST | http://localhost:3000/2015-03-31/functions/app/invocations

Server ready: http://localhost:3000 🚀
```

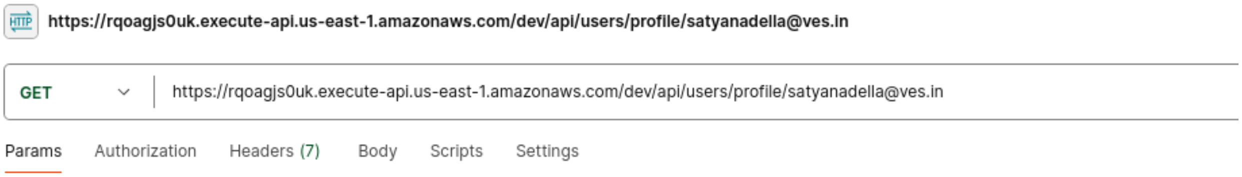
6. Deploy the App
   Once the configuration is ready and application is working fine locally, deploy your app with the following command:
   > **serverless deploy**



```
Deploying "express-backend" to stage "dev" (us-east-1)

✓ Service deployed to stack express-backend-dev (81s)

endpoint: ANY - https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/{proxy+}
functions:
  app: express-backend-dev-app (51 MB)
```

   Our deployment succeeded. Our endpoint base URL is https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/

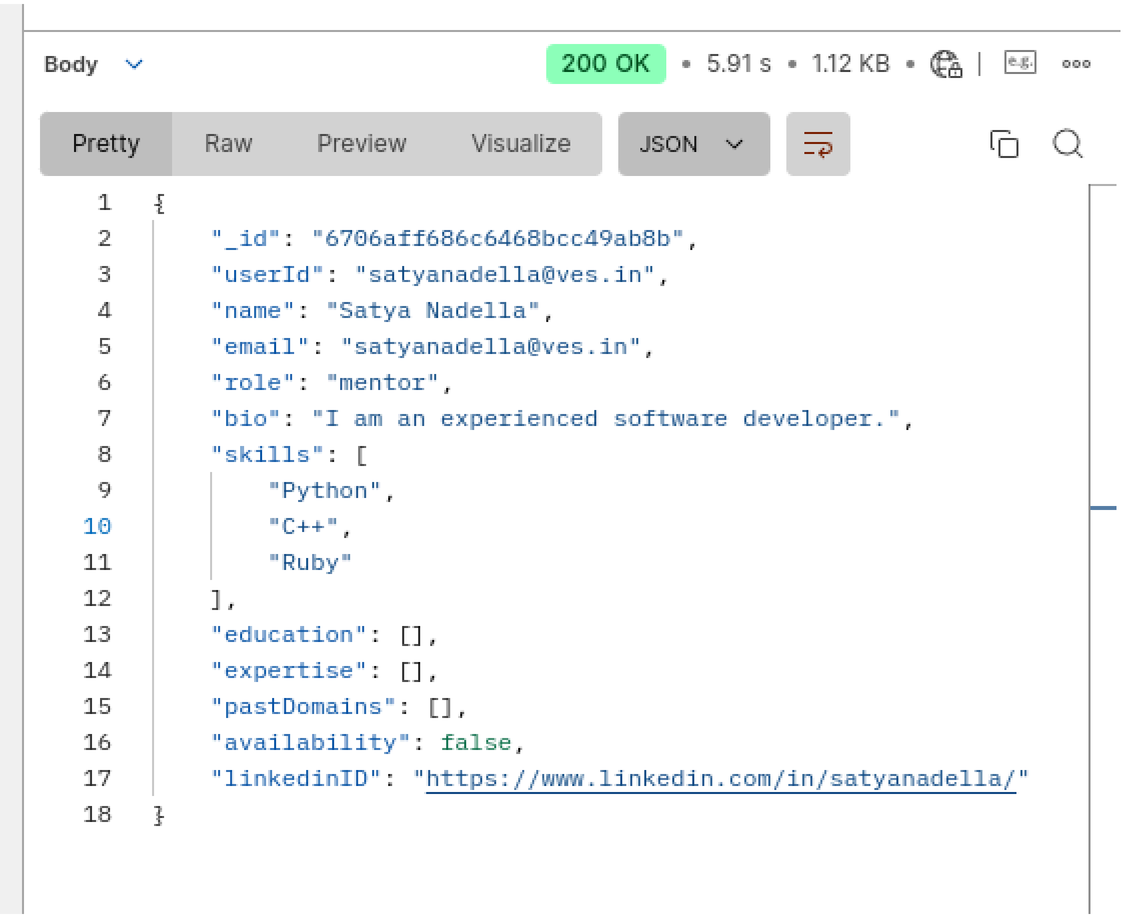
7. Test the deployed application using Postman
   - Now we would fetch the details of existing user by sending **GET** request to https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/api/users/profile/satyanadella@ves.in



https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/api/users/profile/satyanadella@ves.in

GET https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/api/users/profile/satyanadella@ves.in

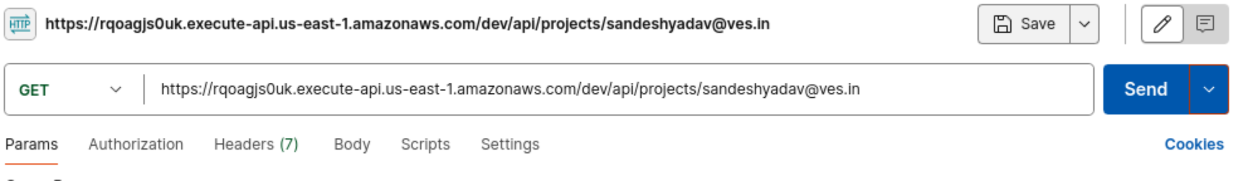
Response from Server



```
{
    "_id": "6706aff686c6468bcc49ab8b",
    "userId": "satyanadella@ves.in",
    "name": "Satya Nadella",
    "email": "satyanadella@ves.in",
    "role": "mentor",
    "bio": "I am an experienced software developer.",
    "skills": [
        "Python",
        "C++",
        "Ruby"
    ],
    "education": [],
    "expertise": [],
    "pastDomains": [],
    "availability": false,
    "linkedinID": "https://www.linkedin.com/in/satyanadella/"
}
```

The server responded with all the details of the user

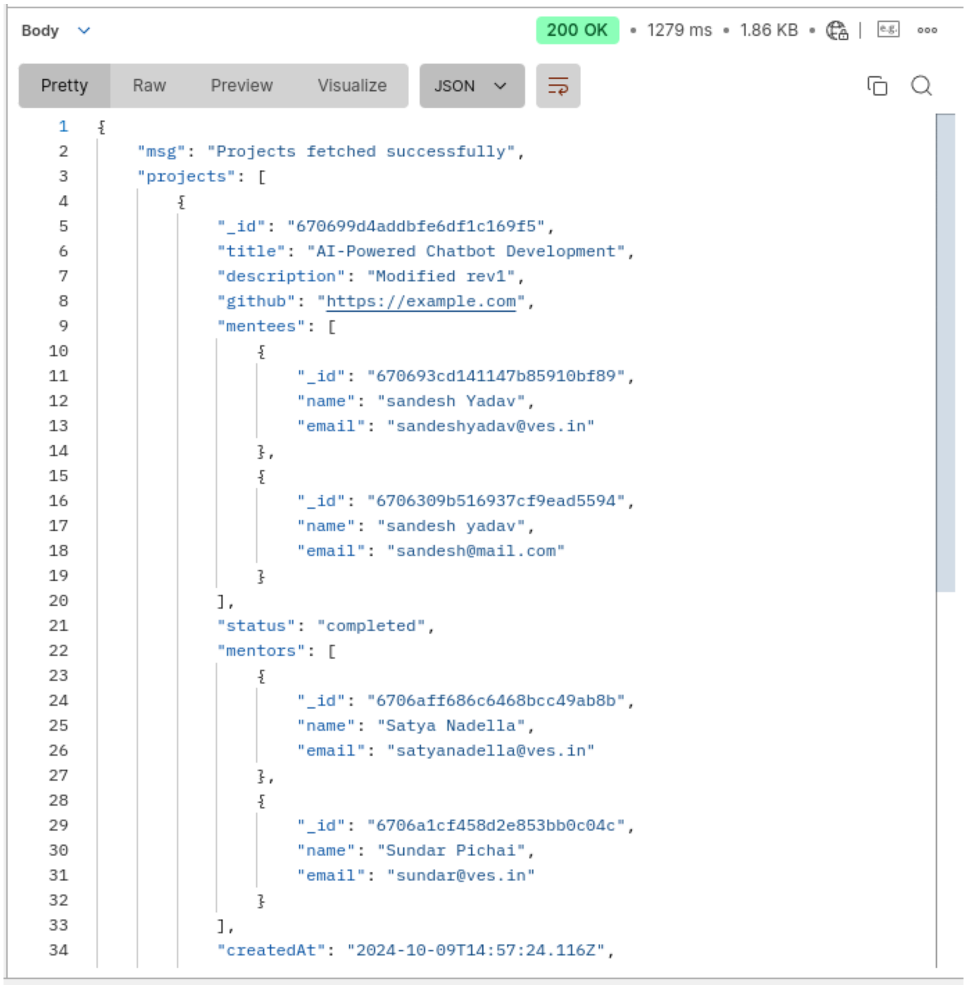
- Now we would fetch all the projects that a user has



https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/api/projects/sandeshyadav@ves.in

GET https://rqoagjs0uk.execute-api.us-east-1.amazonaws.com/dev/api/projects/sandeshyadav@ves.in

Response :



```
{
    "msg": "Projects fetched successfully",
    "projects": [
        {
            "_id": "670699d4addbfe6df1c169f5",
            "title": "AI-Powered Chatbot Development",
            "description": "Modified rev1",
            "github": "https://example.com",
            "mentees": [
                {
                    "_id": "670693cd141147b85910bf89",
                    "name": "sandesh Yadav",
                    "email": "sandeshyadav@ves.in"
                },
                {
                    "_id": "6706309b516937cf9ead5594",
                    "name": "sandesh yadav",
                    "email": "sandesh@mail.com"
                }
            ],
            "status": "completed",
            "mentors": [
                {
                    "_id": "6706aff686c6468bcc49ab8b",
                    "name": "Satya Nadella",
                    "email": "satyanadella@ves.in"
                },
                {
                    "_id": "6706a1cf458d2e853bb0c04c",
                    "name": "Sundar Pichai",
                    "email": "sundar@ves.in"
                }
            ],
            "createdAt": "2024-10-09T14:57:24.116Z",
```

We are to fetch all the projects that a user has

Conclusion :

We successfully deployed a serverless RESTful API as an AWS Lambda function integrated with AWS API Gateway and DynamoDB for handling CRUD operations. We encountered and resolved AWS role permission issues by assigning the necessary permissions. The process involved defining Lambda functions for user retrieval and creation, creating a serverless configuration file to integrate the API Gateway routes, testing locally, and deploying it using the Serverless Framework. Additionally, we deployed the backend server of our third-year project, MentorLink, as a RESTful API supporting all CRUD operations, with minor latency due to non-edge optimization