NAME: MANORATH SUNIL ITAL

CLASS: D15A

ROLL NO: 19

CASE STUDY REPORT

1. Introduction

Case Study Overview:
The chosen case study focuses on building a serverless REST API architecture using Amazon Web Services (AWS) to efficiently manage user data. The primary objective of this project was to design and implement an API that allows for adding and retrieving user details from a cloud-based database. This was achieved by leveraging three core AWS services: Lambda for serverless computing, API Gateway for API management, and DynamoDB for scalable NoSQL storage. The project aimed to demonstrate how serverless solutions can offer scalability, reliability, and cost efficiency while simplifying the deployment and management of APIs.

Serverless computing eliminates the need for traditional server setup, reducing operational overhead and allowing developers to focus on coding rather than infrastructure management. In this case study, the REST API's functionality was restricted to basic CRUD operations: adding a new user and retrieving user details by ID, effectively covering the core elements of a simple user management system.

Key Feature and Application:
The unique feature of this project is its serverless design, achieved through the integration of AWS Lambda, API Gateway, and DynamoDB. This architecture allows the REST API to scale automatically based on the incoming traffic without any manual intervention or additional server management. The use of DynamoDB, a fully managed NoSQL database, ensures that user data is stored efficiently and retrieved quickly, meeting the demands of high-throughput applications.

From a practical perspective, the serverless REST API serves as a foundational structure for real-world applications that need to manage user data or similar information. It can be applied in various scenarios, such as:

1. **User Management Systems:** The API could be used in a web or mobile application to register new users, manage profiles, and retrieve user information securely and efficiently.
2. **Content Management Systems (CMS):** With minor modifications, the serverless architecture can handle content-related data for websites or platforms, providing functionalities like adding, updating, or retrieving content entries.

3. **IoT Applications:** The flexibility and scalability of serverless architecture make it suitable for IoT-based applications, where sensor data or user activity needs to be stored and accessed frequently.

Additionally, this approach reduces infrastructure management costs and complexity, as developers only pay for actual usage of Lambda functions, making it an economical solution, especially for startups and small-scale applications.

2. Step-by-Step Explanation

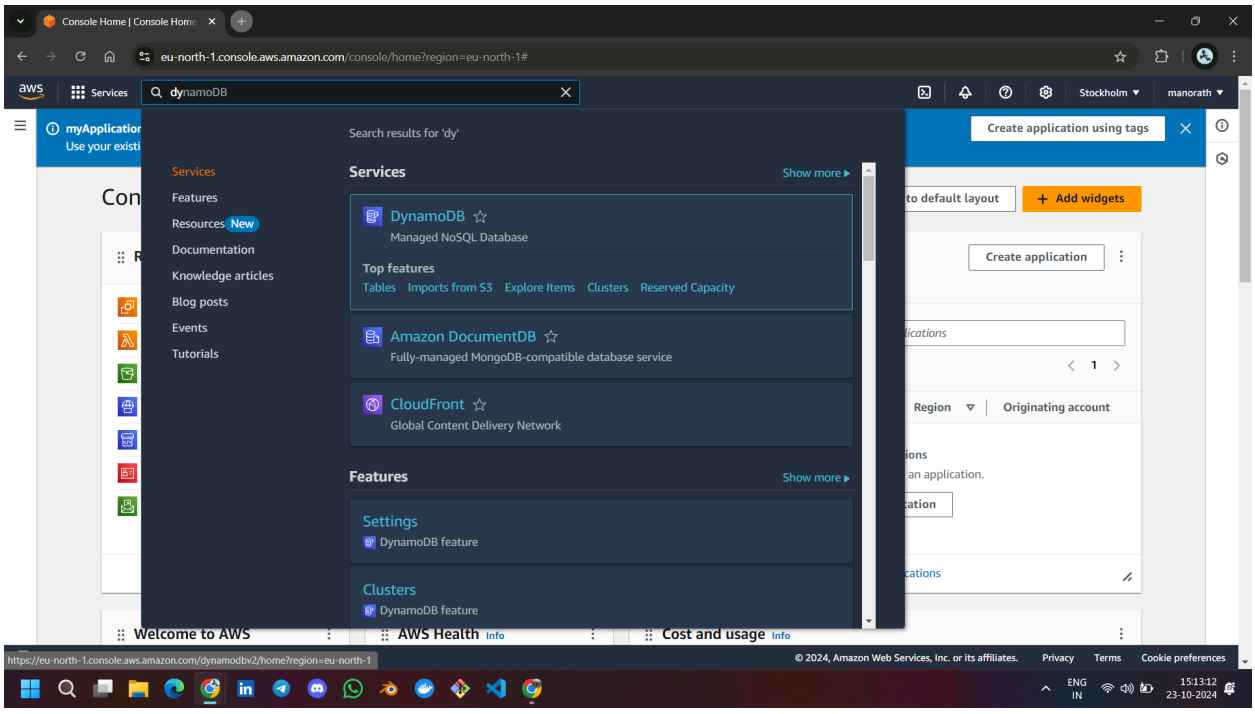
Building a serverless REST API using AWS Lambda, API Gateway, and DynamoDB:

Prerequisites:

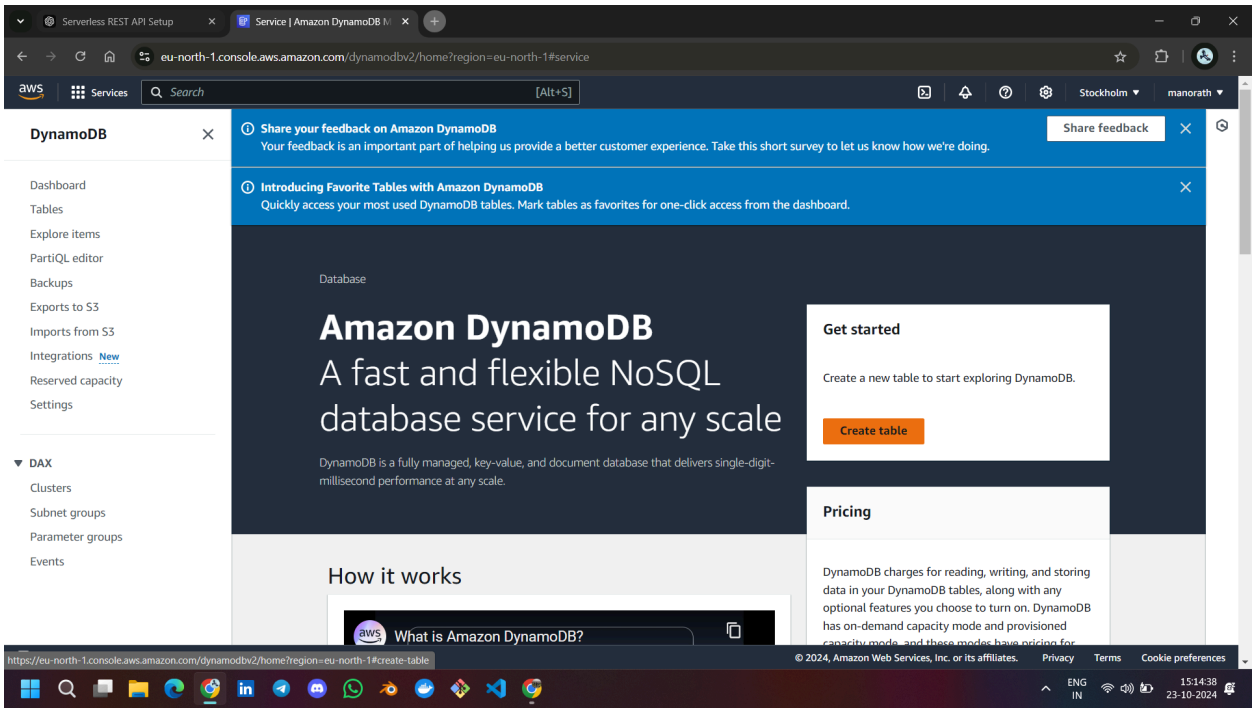
- **AWS Account** with access to AWS Management Console.
- **Basic knowledge of AWS services** such as Lambda, API Gateway, and DynamoDB.
- **Node.js or Python** installed on your local machine (for writing Lambda functions).
- **Postman or curl** for testing.

Step 1: Create a DynamoDB Table

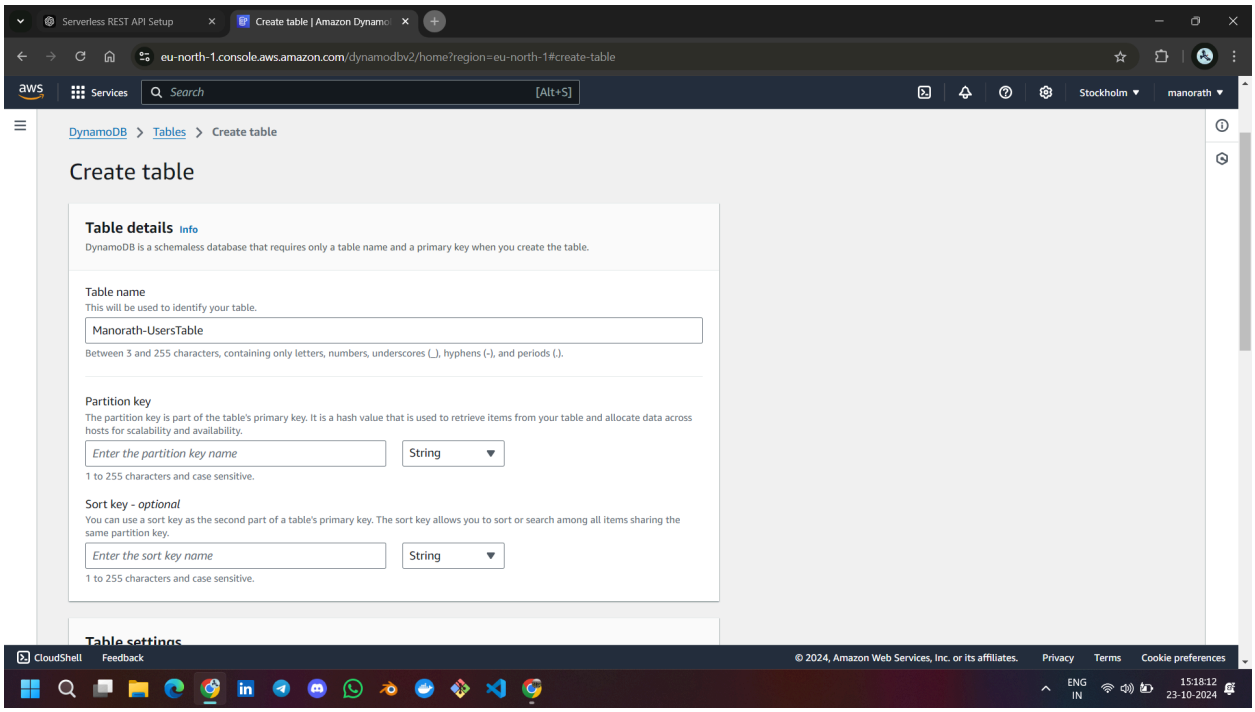
1. **Go to AWS Console** and open the DynamoDB service.



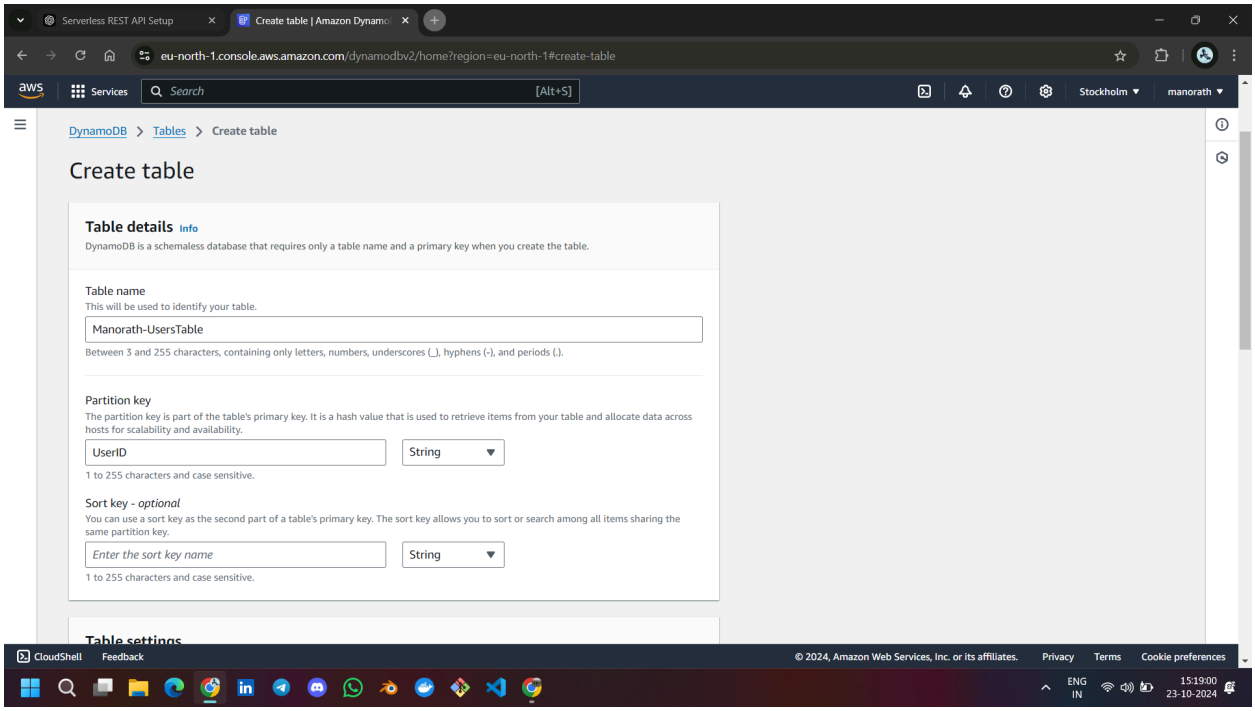
2. Click on **Create Table**.



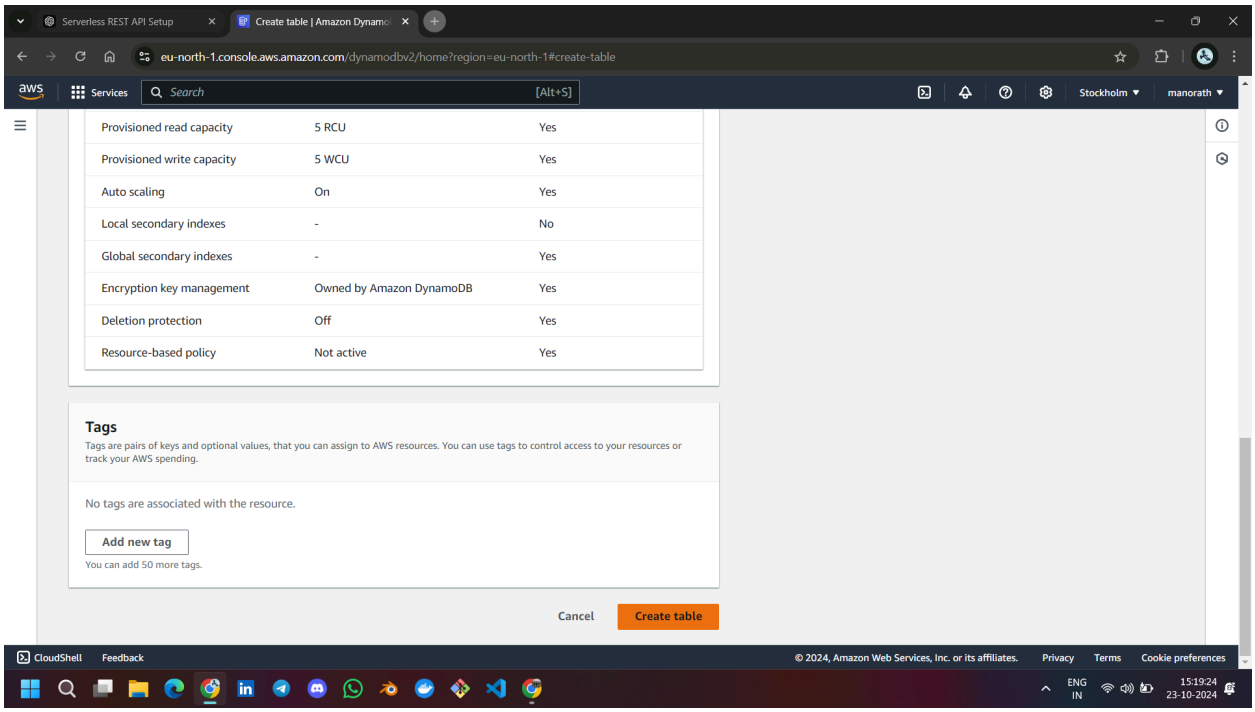
3. Enter a **Table Name** (e.g., `UsersTable`).



4. Enter a **Primary Key** (e.g., `UserID` as a `String`).



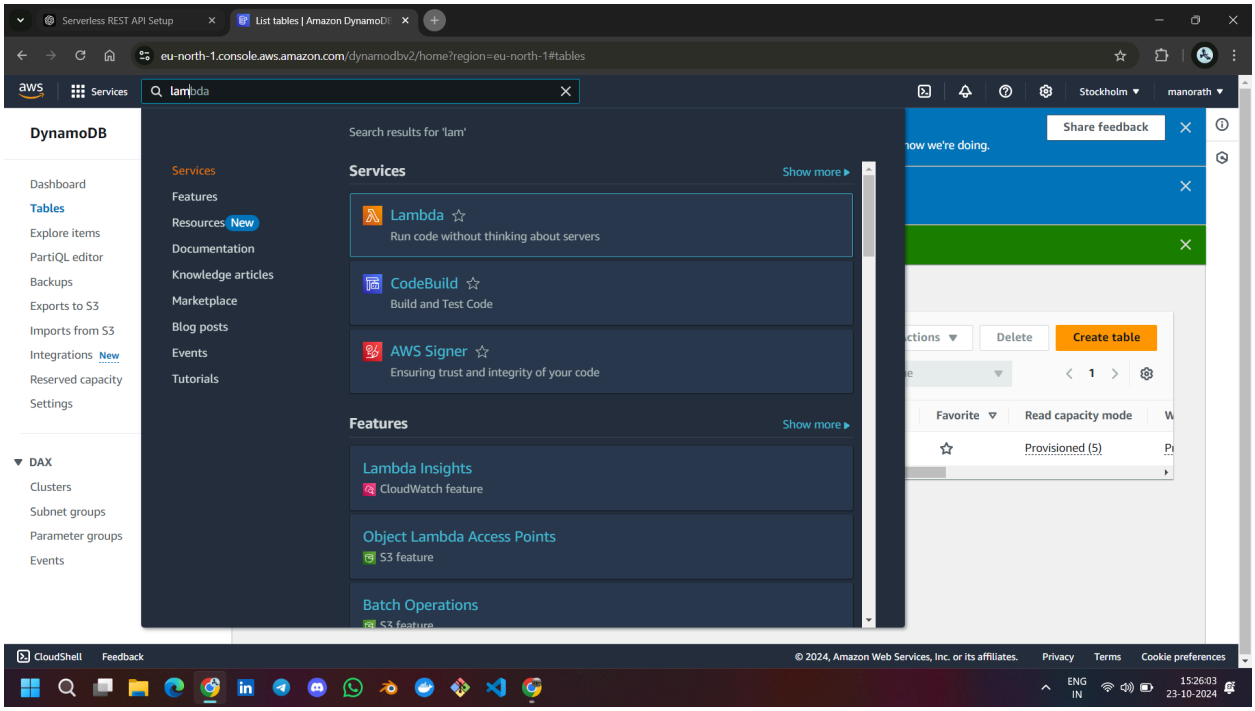
5. Leave the rest of the settings as default and click **Create**.

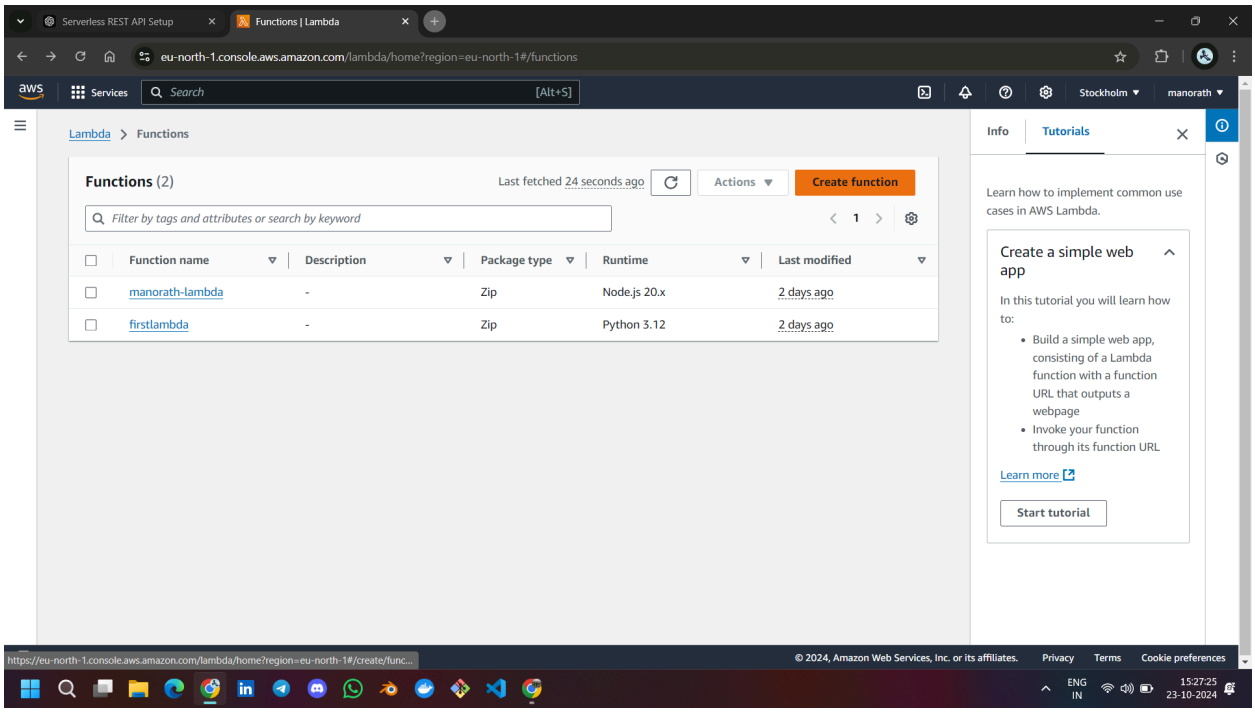


Step 2: Create Lambda Functions

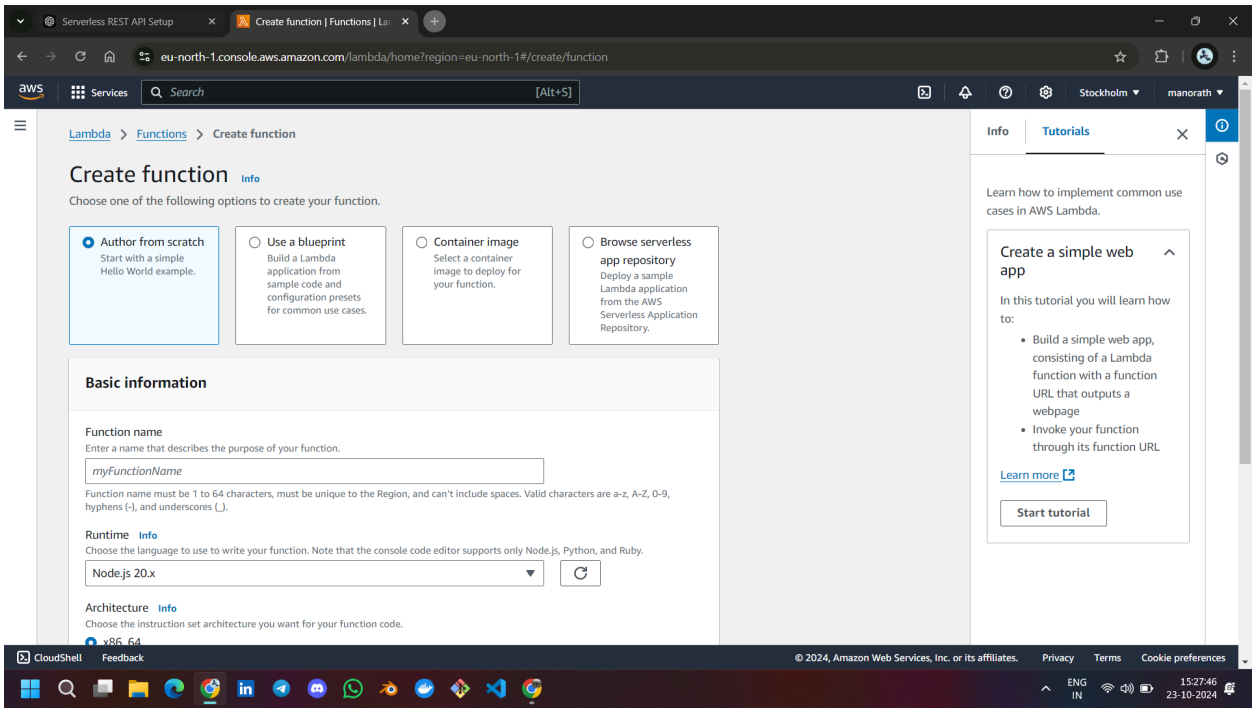
Lambda Function 1: Add a User

1. Go to the **AWS Lambda** service and click on **Create Function**.

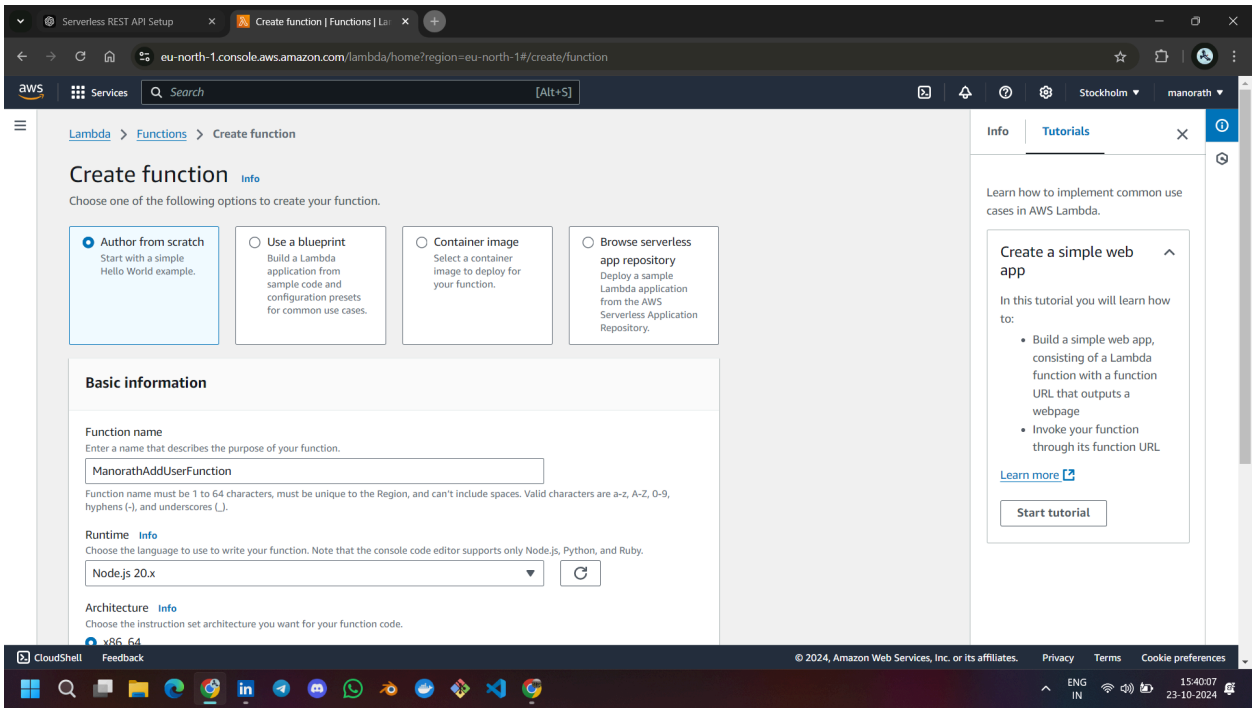




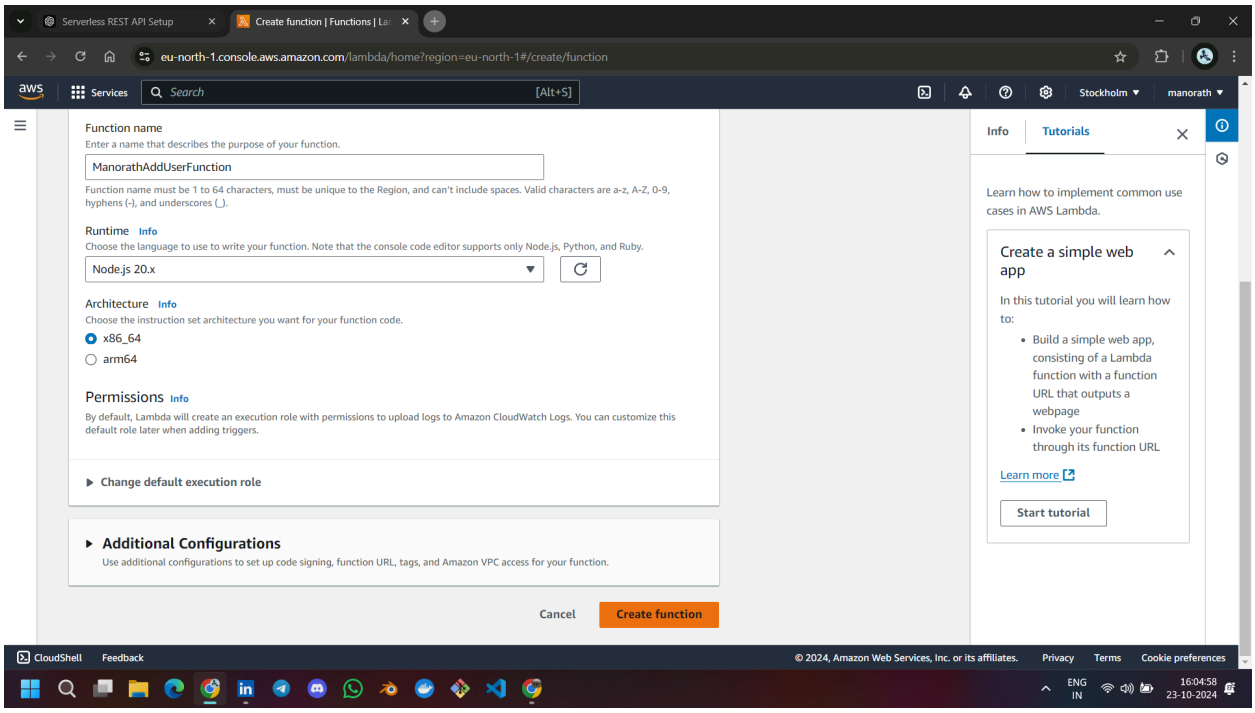
2. Choose **Author from scratch**.



3. Give it a name (e.g., `AddUserFunction`) and select **Runtime** (e.g., `Node.js 14.x`).



4. Click on **Create Function**.

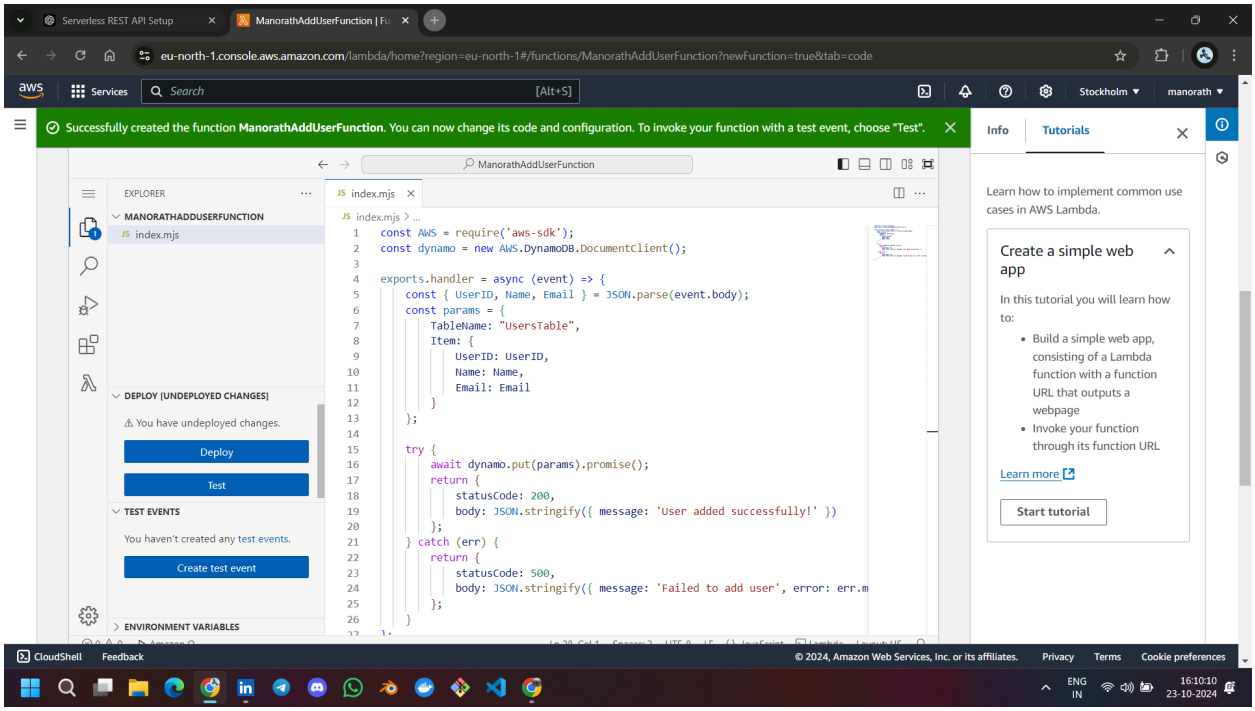


5.Inside the function editor, add the following code to insert a user into DynamoDB:
javascript
Copy code

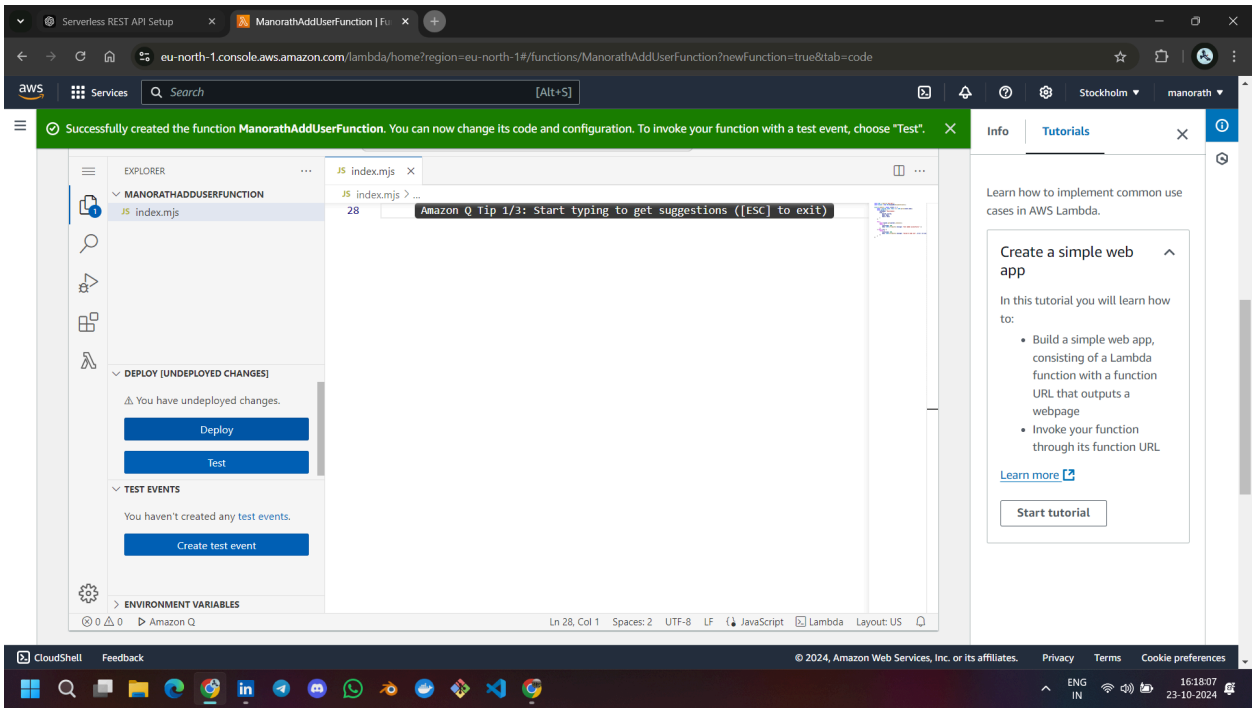
```
const AWS = require('aws-sdk');
const dynamo = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {
    const { UserID, Name, Email } = JSON.parse(event.body);
    const params = {
        TableName: "UsersTable",
        Item: {
            UserID: UserID,
            Name: Name,
            Email: Email
        }
    };

    try {
        await dynamo.put(params).promise();
        return {
```

```
            statusCode: 200,
            body: JSON.stringify({ message: 'User added successfully!'
})
        };
    } catch (err) {
        return {
            statusCode: 500,
            body: JSON.stringify({ message: 'Failed to add user',
error: err.message })
        };
    }
};
```

5.

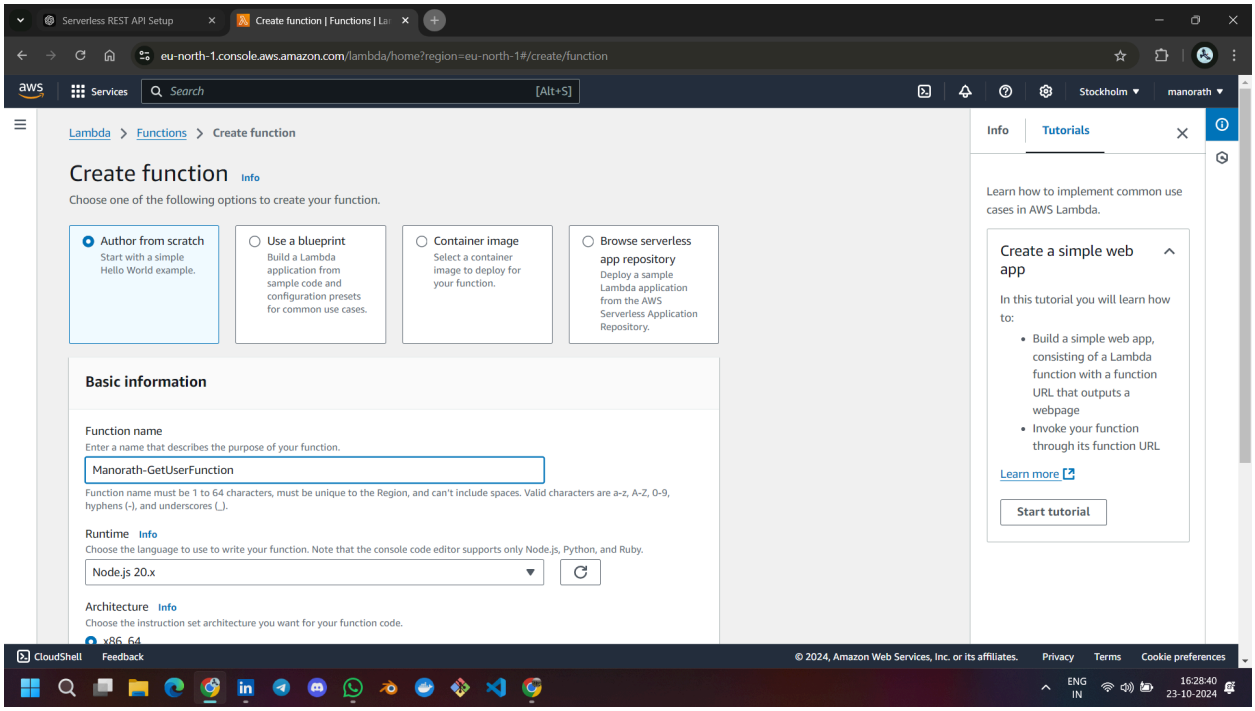


6. Click **Deploy** to save the function.



Lambda Function 2: Get User by ID

1. Create another Lambda function (e.g., `GetUserFunction`), following the same steps.

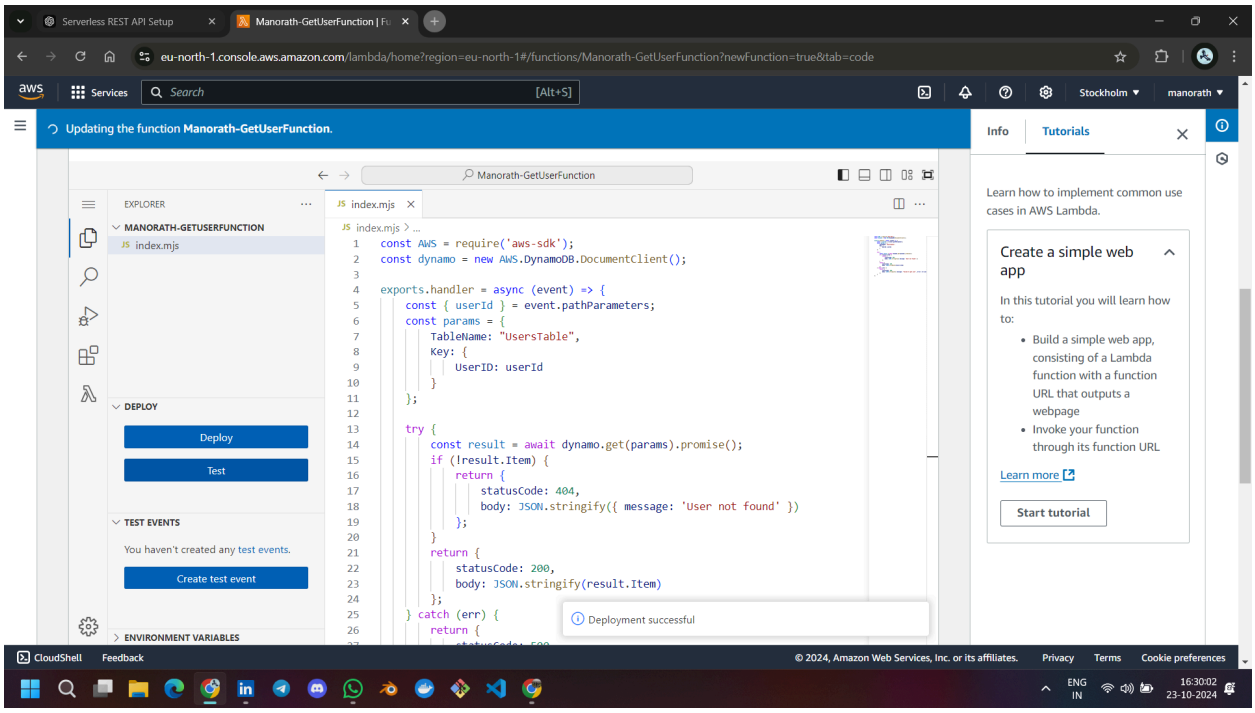


MANORATH ITAL D15A/19

Add the following code to retrieve a user from DynamoDB:
javascript
Copy code

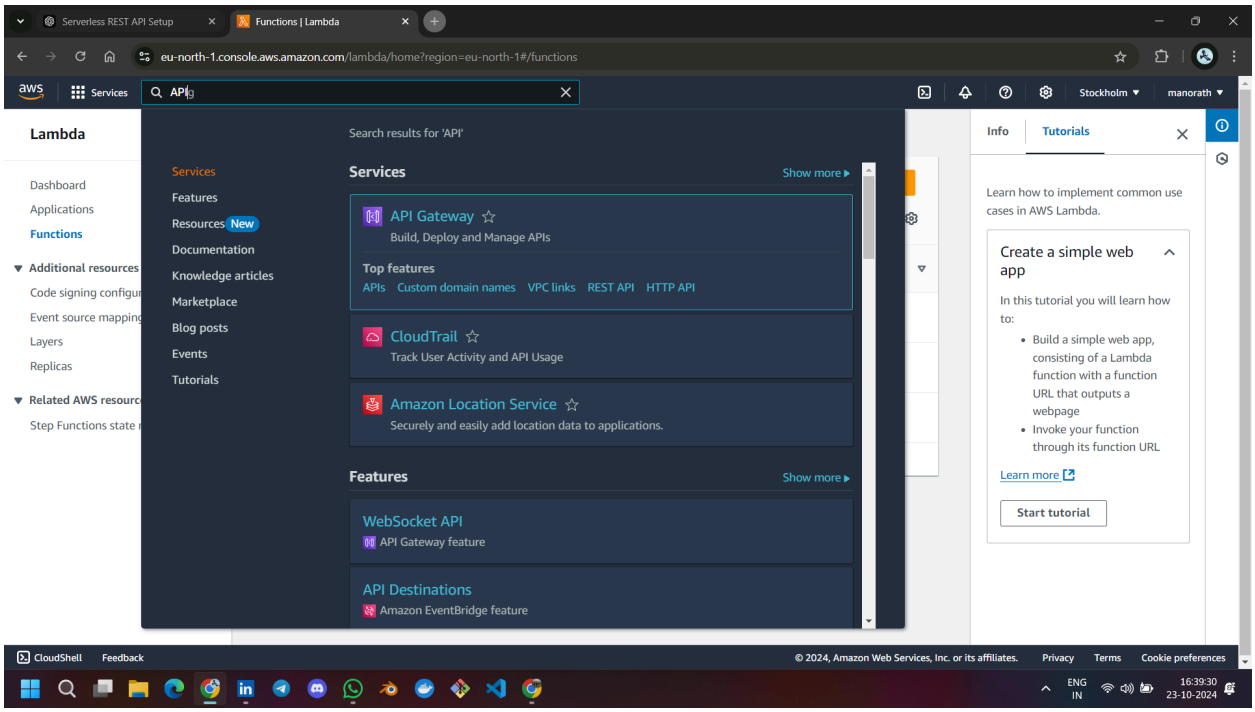
```
const AWS = require('aws-sdk');
const dynamo = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {
    const { userId } = event.pathParameters;
    const params = {
        TableName: "UsersTable",
        Key: {
            UserID: userId
        }
    };

    try {
        const result = await dynamo.get(params).promise();
        if (!result.Item) {
            return {
                statusCode: 404,
                body: JSON.stringify({ message: 'User not found' })
            };
        }
        return {
            statusCode: 200,
            body: JSON.stringify(result.Item)
        };
    } catch (err) {
        return {
            statusCode: 500,
            body: JSON.stringify({ message: 'Failed to get user',
error: err.message })
        };
    }
};
```

2. Click **Deploy** to save the function.

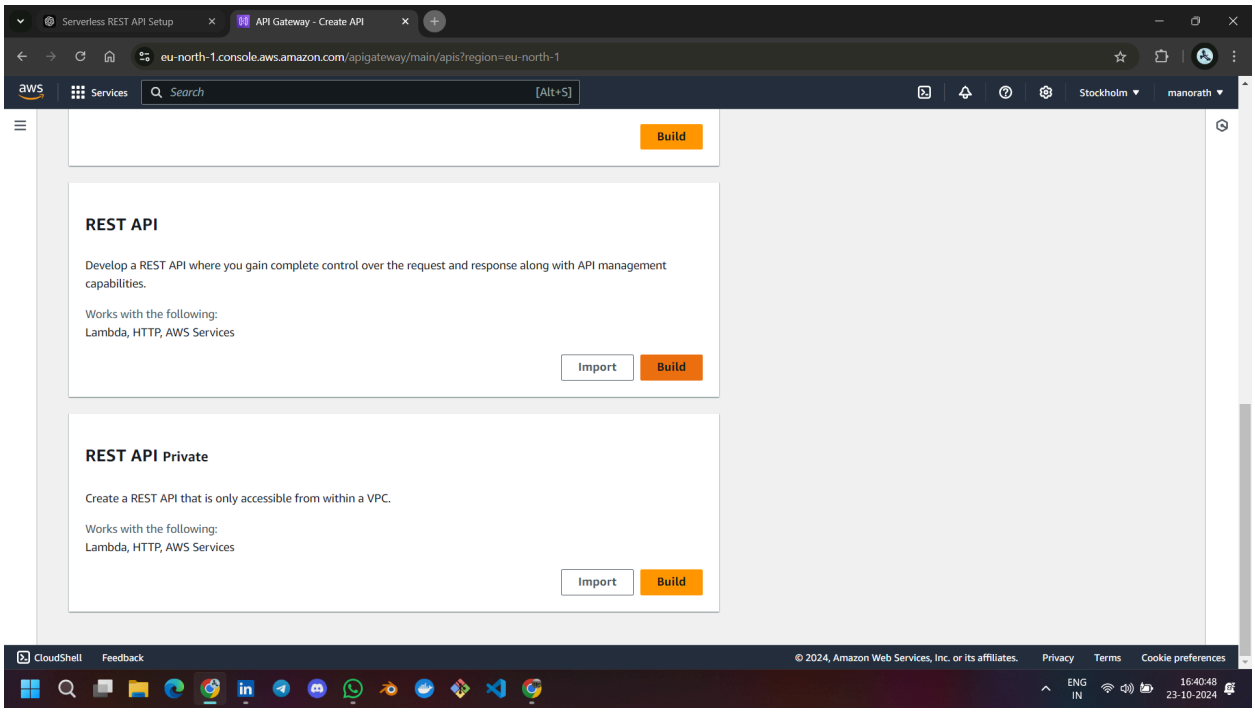


Step 3: Create an API Gateway

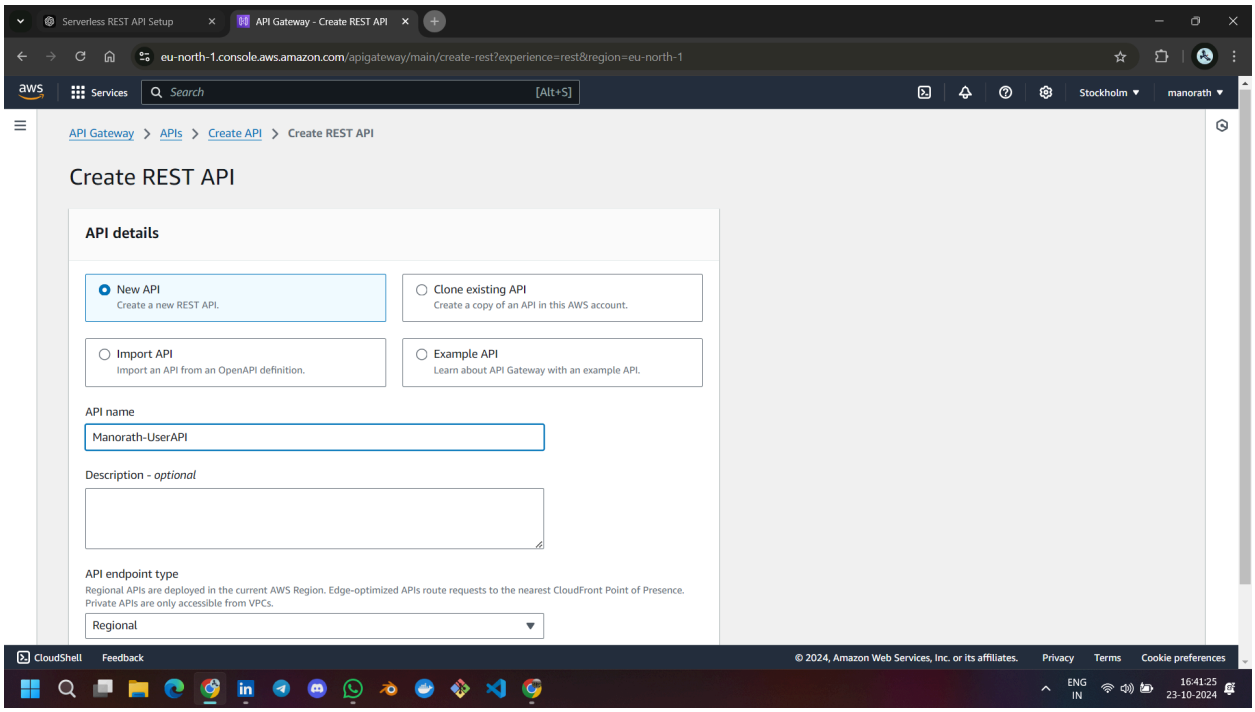
1. Go to **API Gateway** in the AWS Console and choose **Create API**.



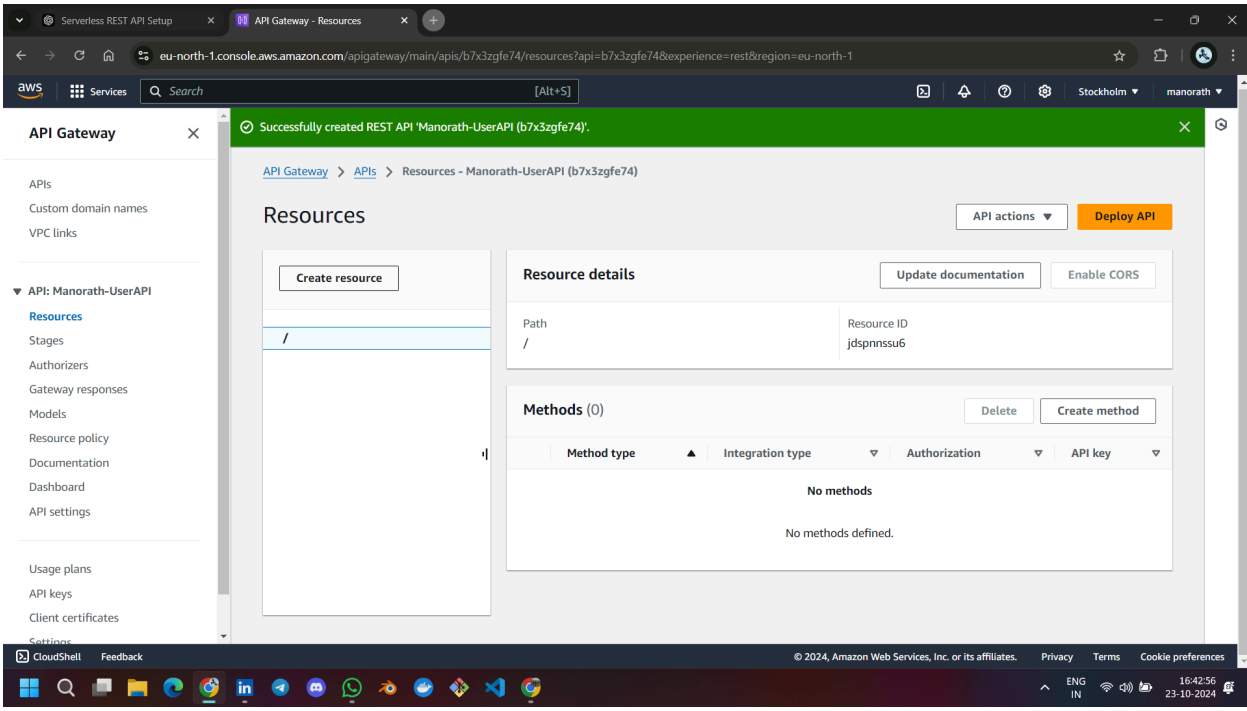
2. Choose **REST API** and then **Build**.



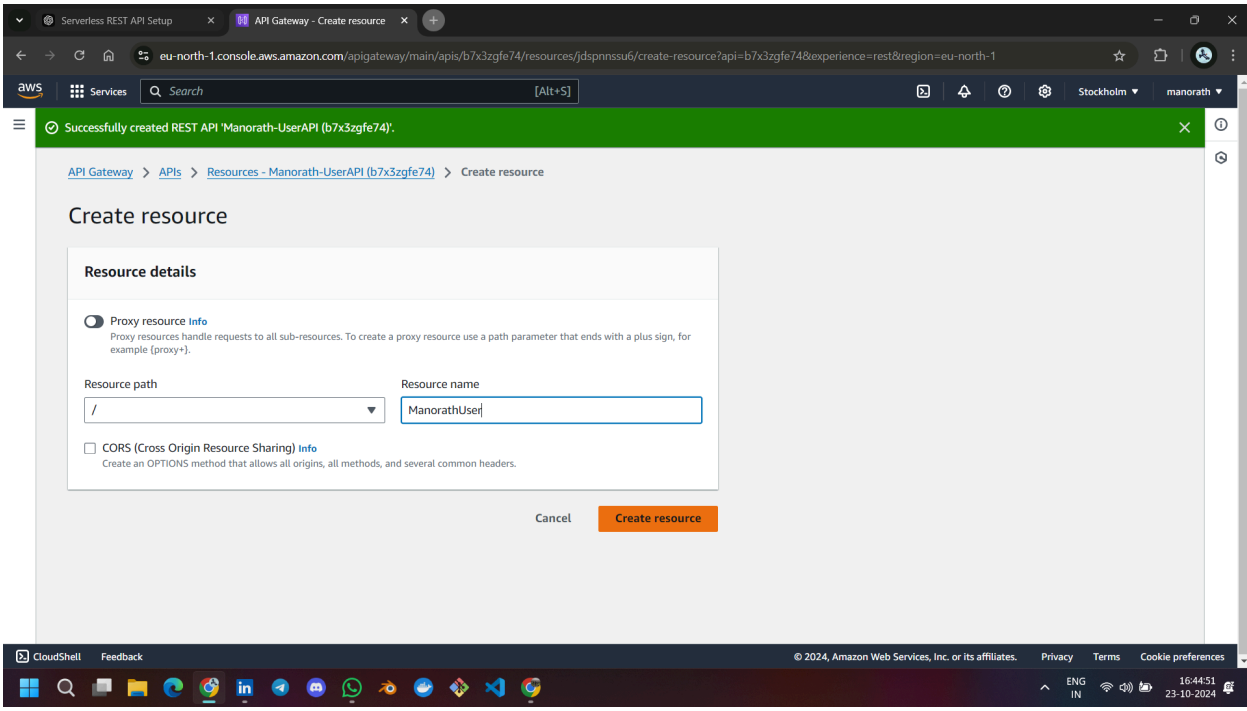
3. **Name your API** (e.g., `UserAPI`) and click **Create API**.



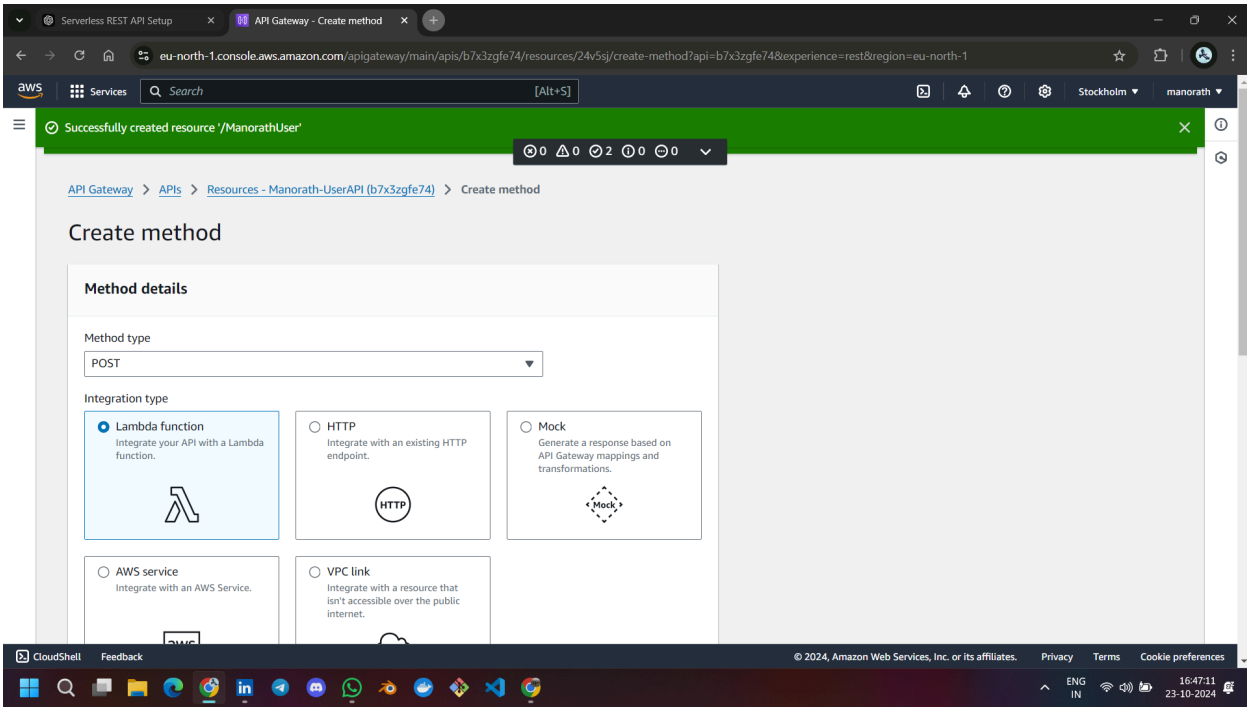
4. **Create Resource**:
   - Click on **Actions → Create Resource**.



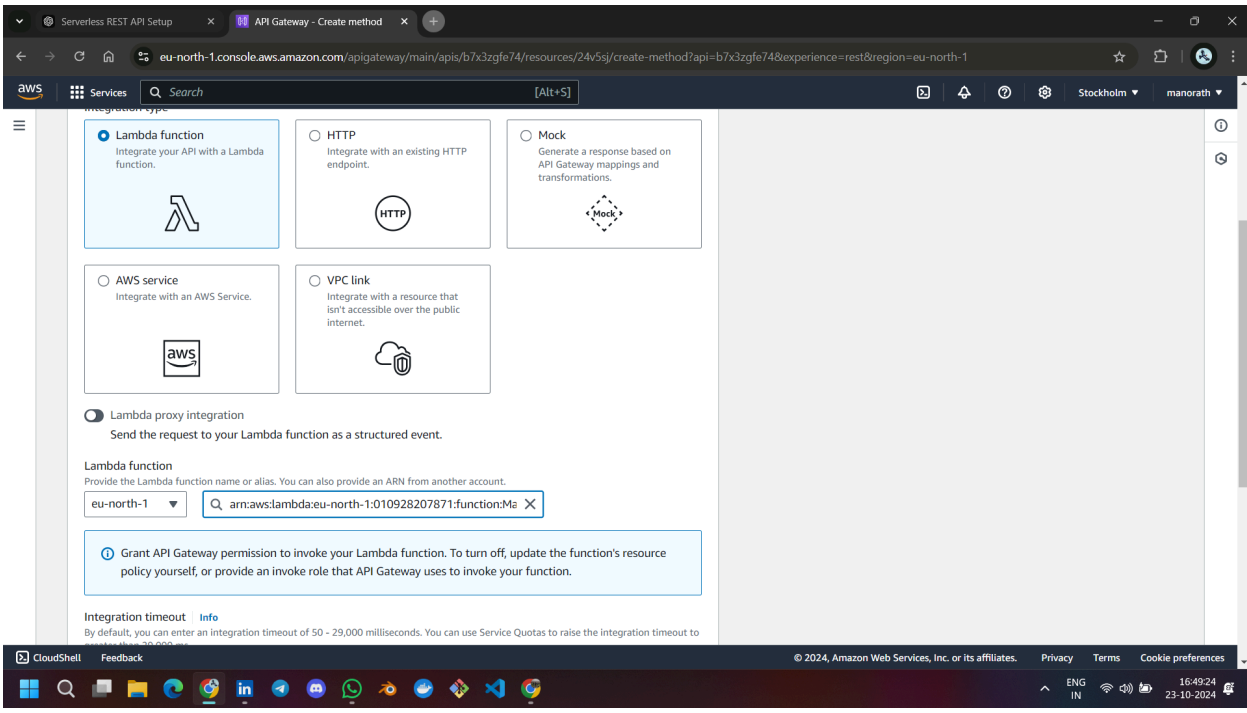
   - Give it a name (e.g., `user`) and click **Create Resource**.



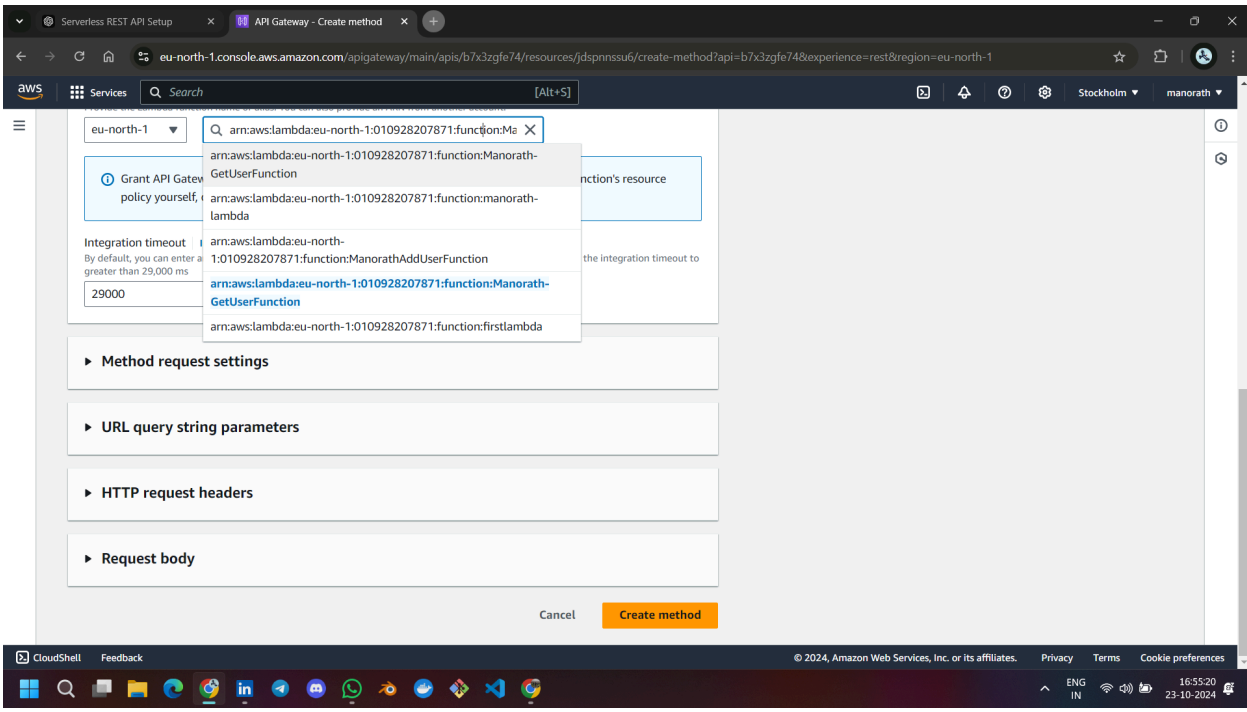
5. **Setup Methods**:
    - Click on **Actions** → **Create Method**.
    - Choose **POST** for adding a user, and then click on the checkmark.



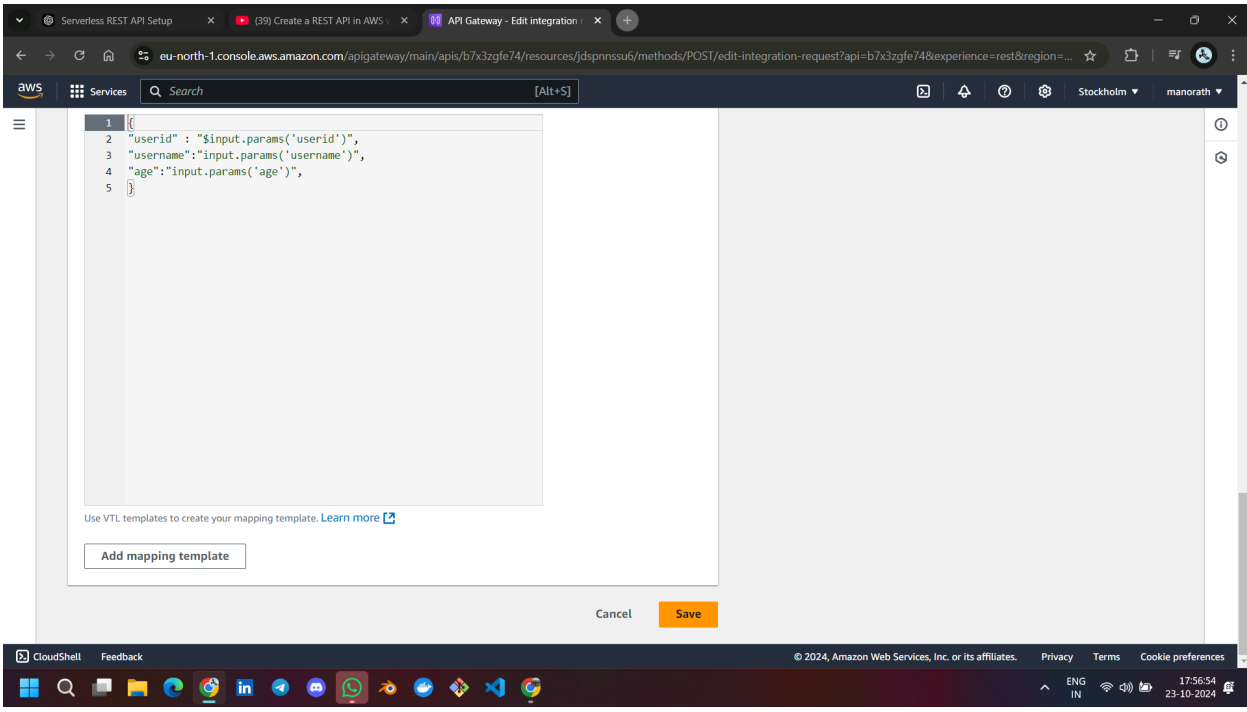
    - In the setup screen, select **Lambda Function** and enter the Lambda function name (`AddUserFunction`). Click **Save**.



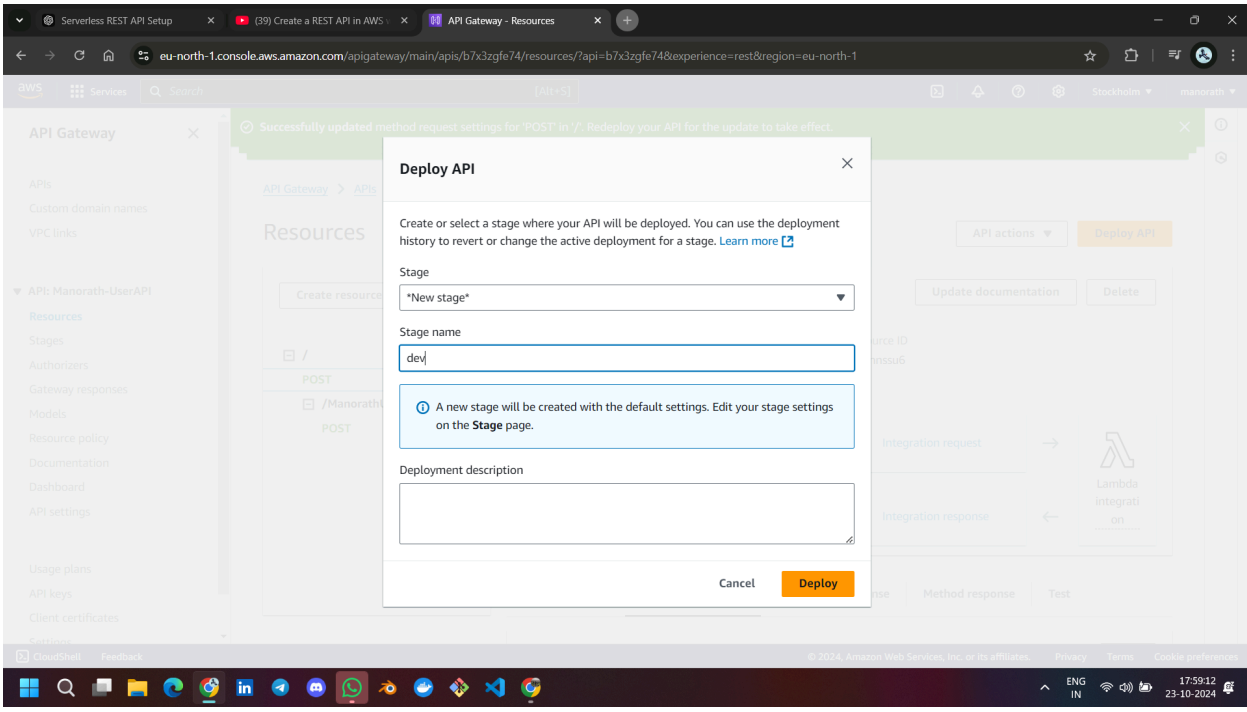
- Repeat this process to create a **GET method** for retrieving a user by ID. Choose **GET** and enter the Lambda function (`GetUserFunction`).



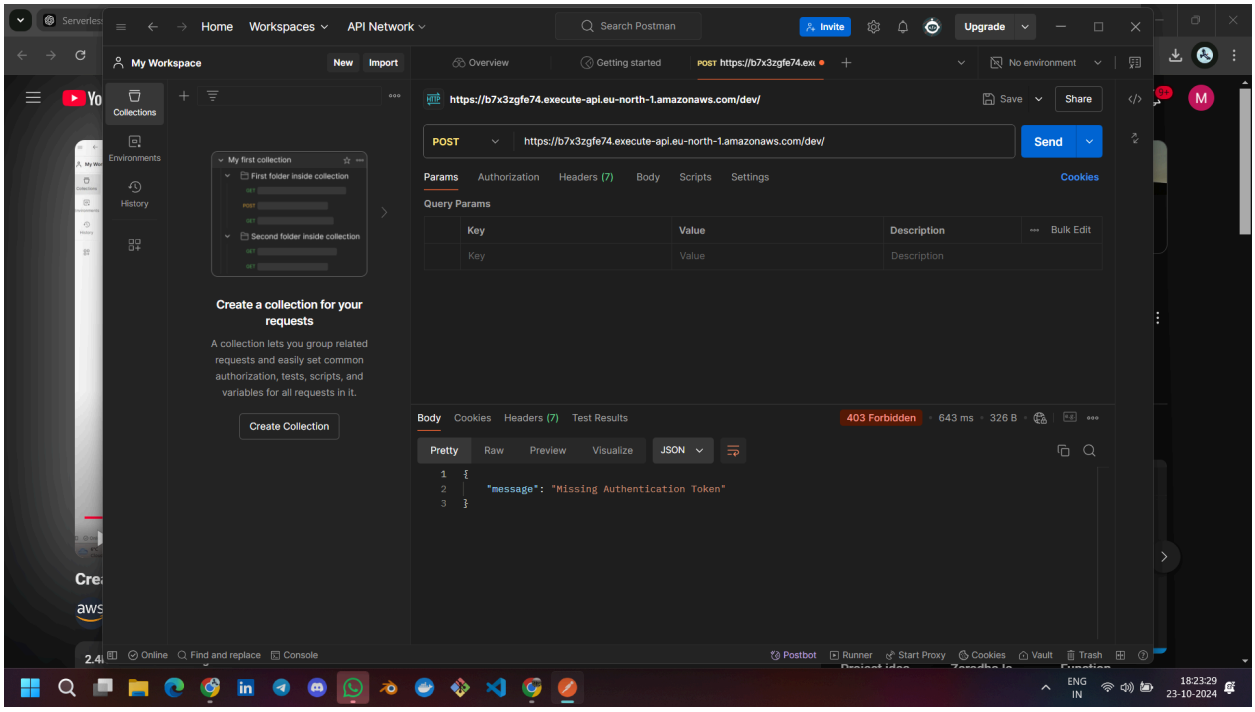
6. **Configure Path Parameter** for the GET method:
    - Click on the **GET method** under your resource.
    - In the **Resource Path**, add `{userId}` to capture the user ID from the URL.



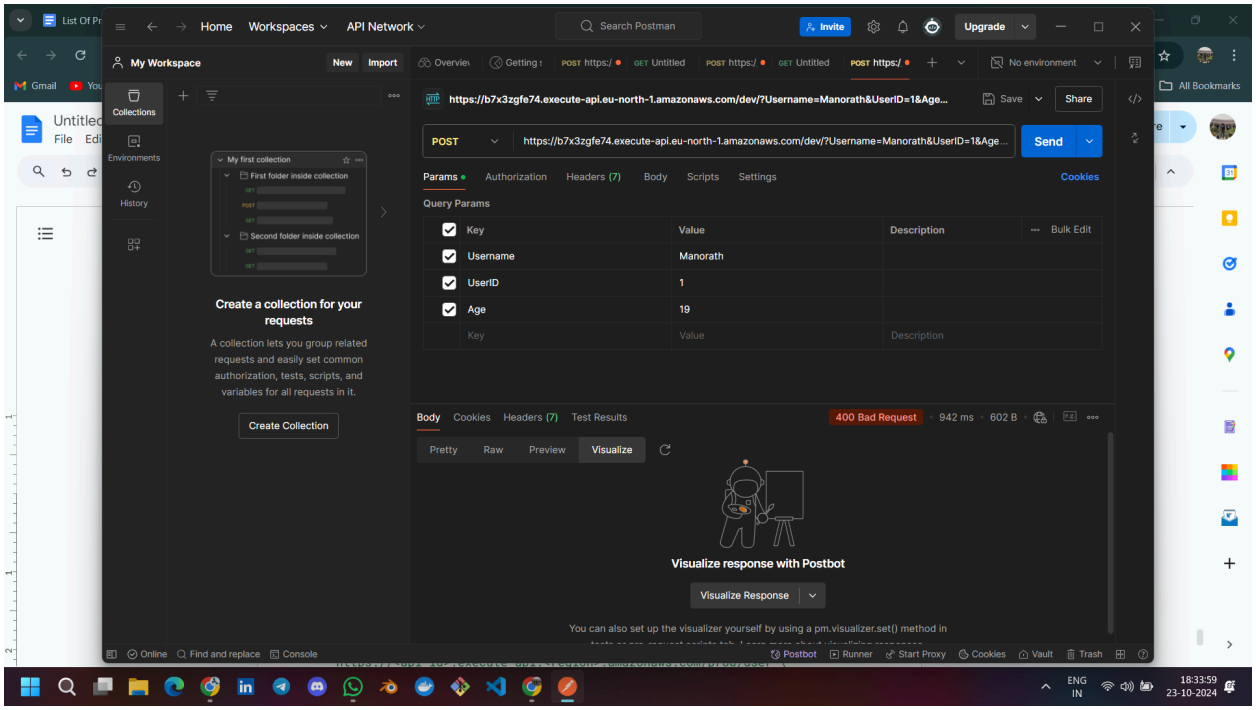
7. **Deploy the API**:
    - Click on **Actions** → **Deploy API**.
    - Create a new stage (e.g., `prod`) and click **Deploy**.

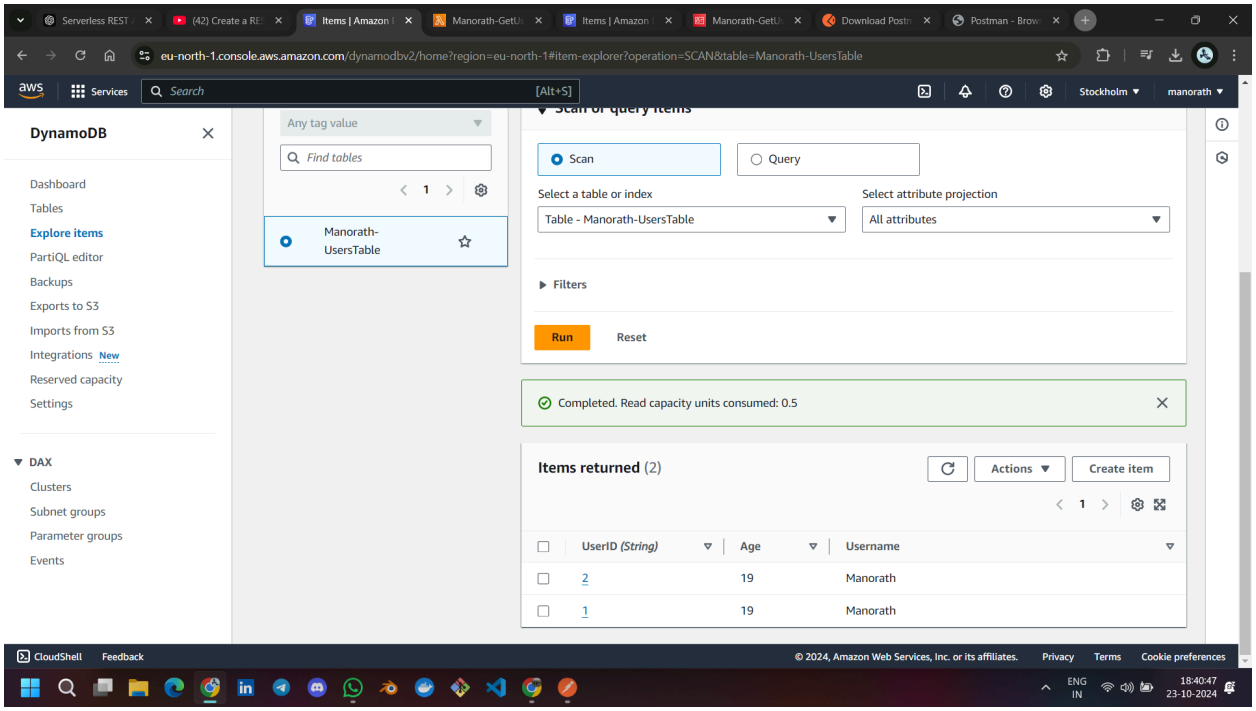


Step 4: Test the API



MANORATH ITAL D15A/19





Summary:

- **DynamoDB Table** for user data storage.
- **Lambda Functions** for adding and retrieving users.
- **API Gateway** setup with methods POST (for adding) and GET (for retrieving).

- **Testing** using curl or Postman.

This setup creates a fully serverless architecture using AWS Lambda, API Gateway, and DynamoDB

Conclusion

Building a serverless REST API using AWS Lambda, API Gateway, and DynamoDB provides an efficient and scalable solution for managing user data. This architecture eliminates the need for managing servers, offering cost-effective scaling and automated availability. By implementing this solution, you leverage AWS's powerful cloud services to create a secure and reliable API for user management. This approach is highly suitable for applications that require high availability and a pay-per-use pricing model.

The project demonstrated how to set up a DynamoDB table, write Lambda functions to handle data operations, and configure an API Gateway to manage and route HTTP requests. The step-by-step integration of these services results in a completely serverless and easily manageable REST API infrastructure, ideal for applications that need a quick deployment without worrying about underlying infrastructure.

Challenges Faced

1. **Configuring Path Parameters in API Gateway**: One key challenge was properly setting up and capturing path parameters in API Gateway to pass them dynamically to Lambda functions. This required careful configuration of resource paths and mapping templates to ensure accurate parameter passing.
2. **Handling CORS Issues**: During testing, handling Cross-Origin Resource Sharing (CORS) became a hurdle. The API responses needed to be configured properly in API Gateway to allow external applications (like Postman or web apps) to communicate without CORS errors.