SJSU SAN JOSÉ STATE UNIVERSITY

Assignment 2

Building serverless application

[CMPE272 - Enterprise Software Platform]

Author,

Yash Bharatbhai Savani
M.S. Software Engineering
SJSU ID: 017581122

Guidance by,

Prof. Rakesh Ranjan
Director of IBM
Software Support

Objective:

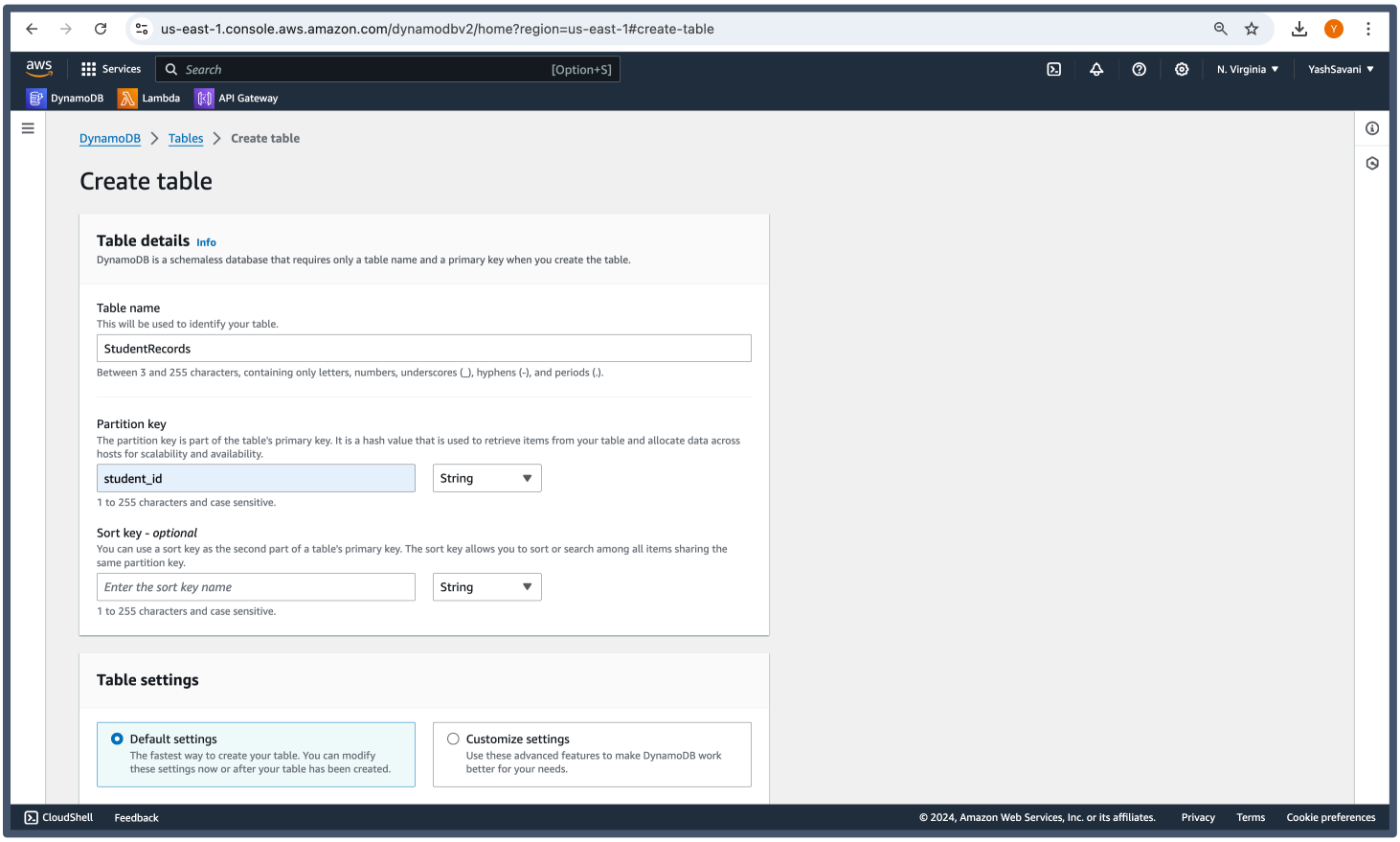
In this assignment, you will create a simple serverless web application using **AWS Lambda** and **Amazon DynamoDB** as the database service. The goal is to understand how to:

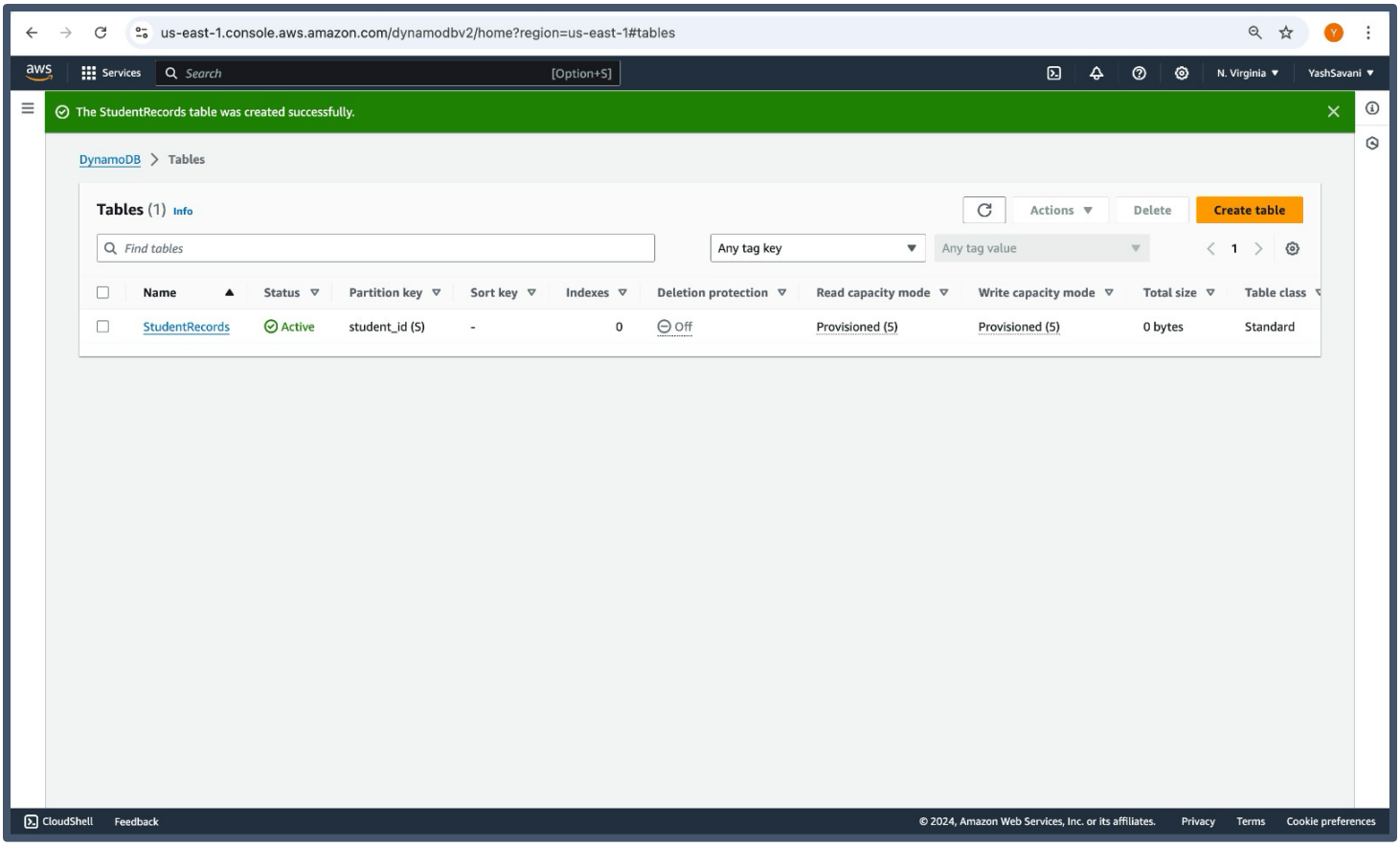
1. Trigger a Lambda function using an API Gateway.
2. Interact with DynamoDB to perform basic CRUD operations (Create, Read, Update, Delete).
3. Deploy and test the application in the AWS environment.

Steps:

1. Setting Up the DynamoDB Table

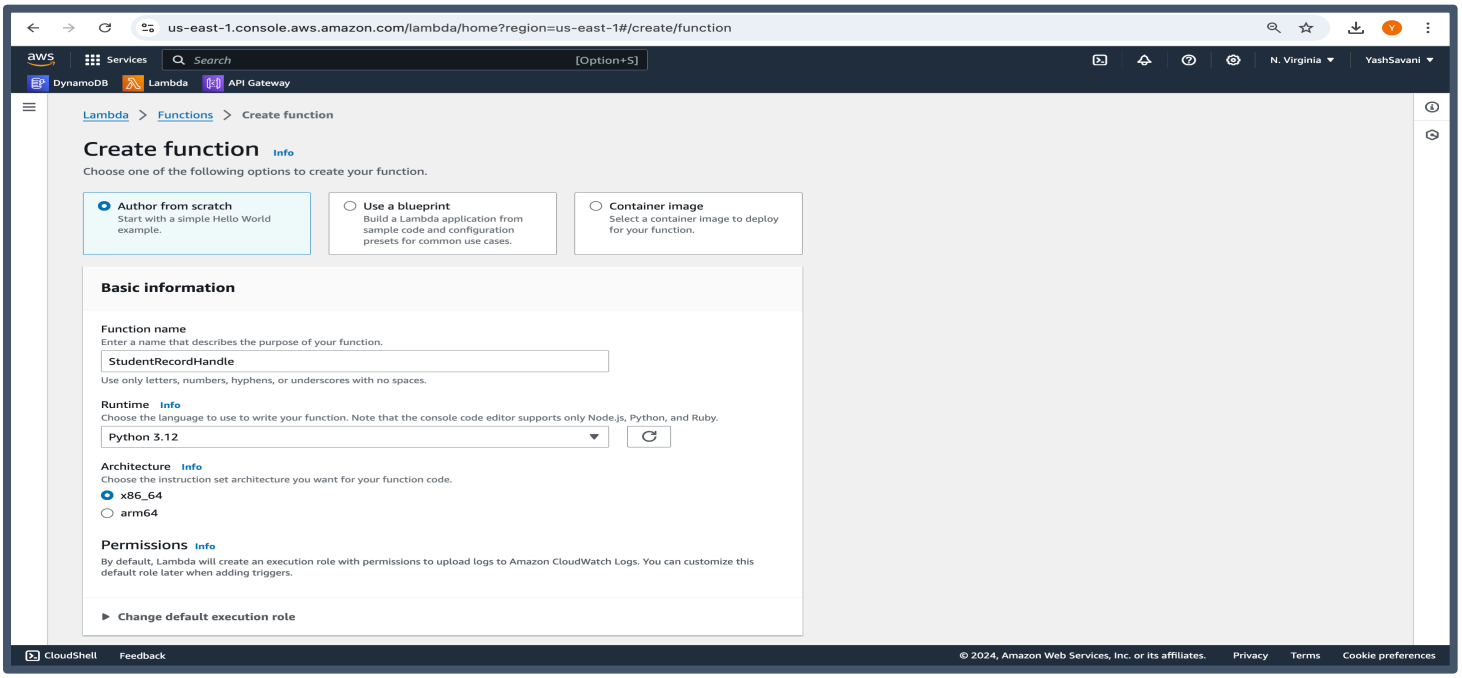
- Go to the AWS Management Console and navigate to **DynamoDB**.
- Create a new table:
    - **Table Name**: StudentRecords
    - **Primary Key**: student_id (String)
- After the table is created, note down the table name.

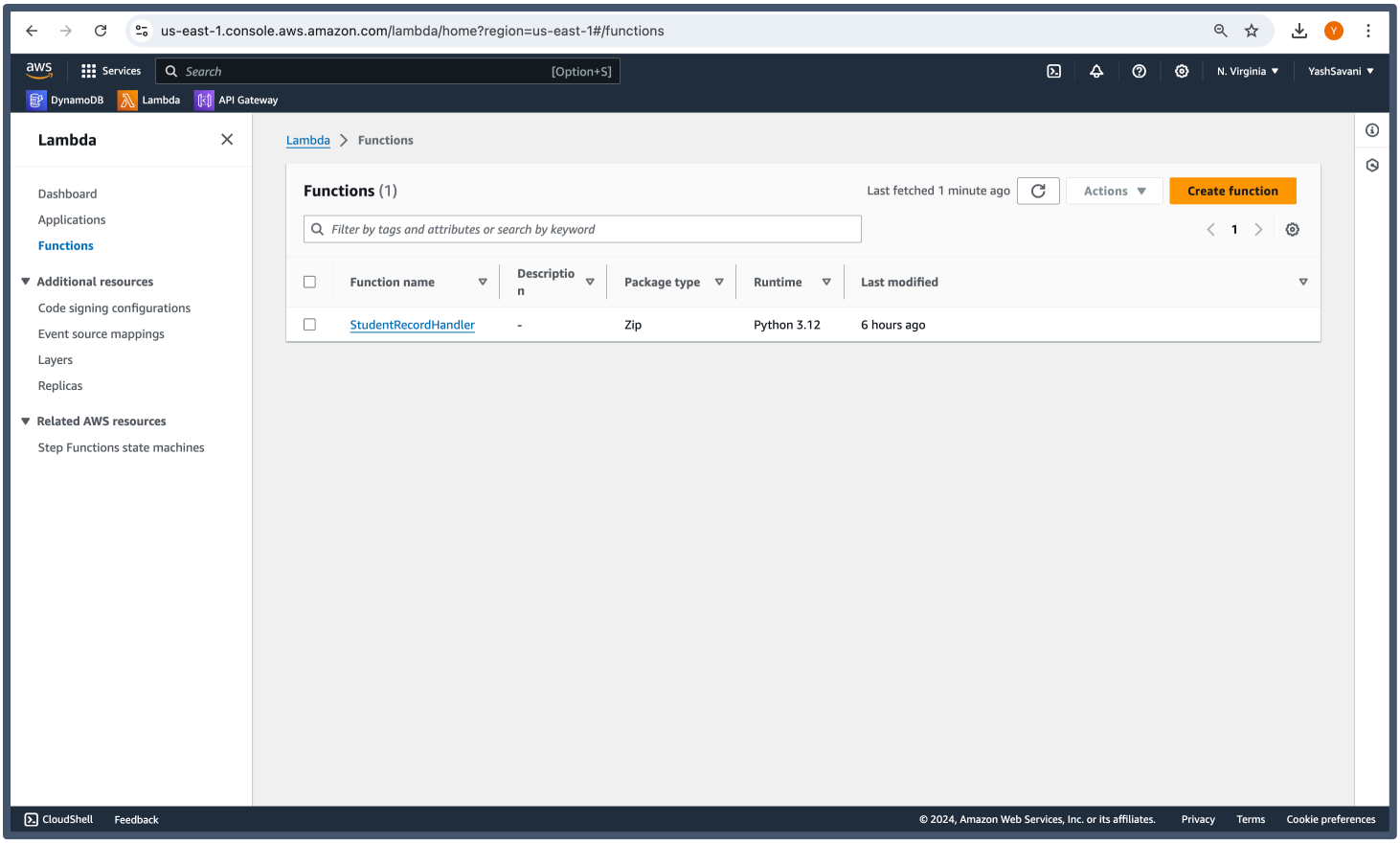


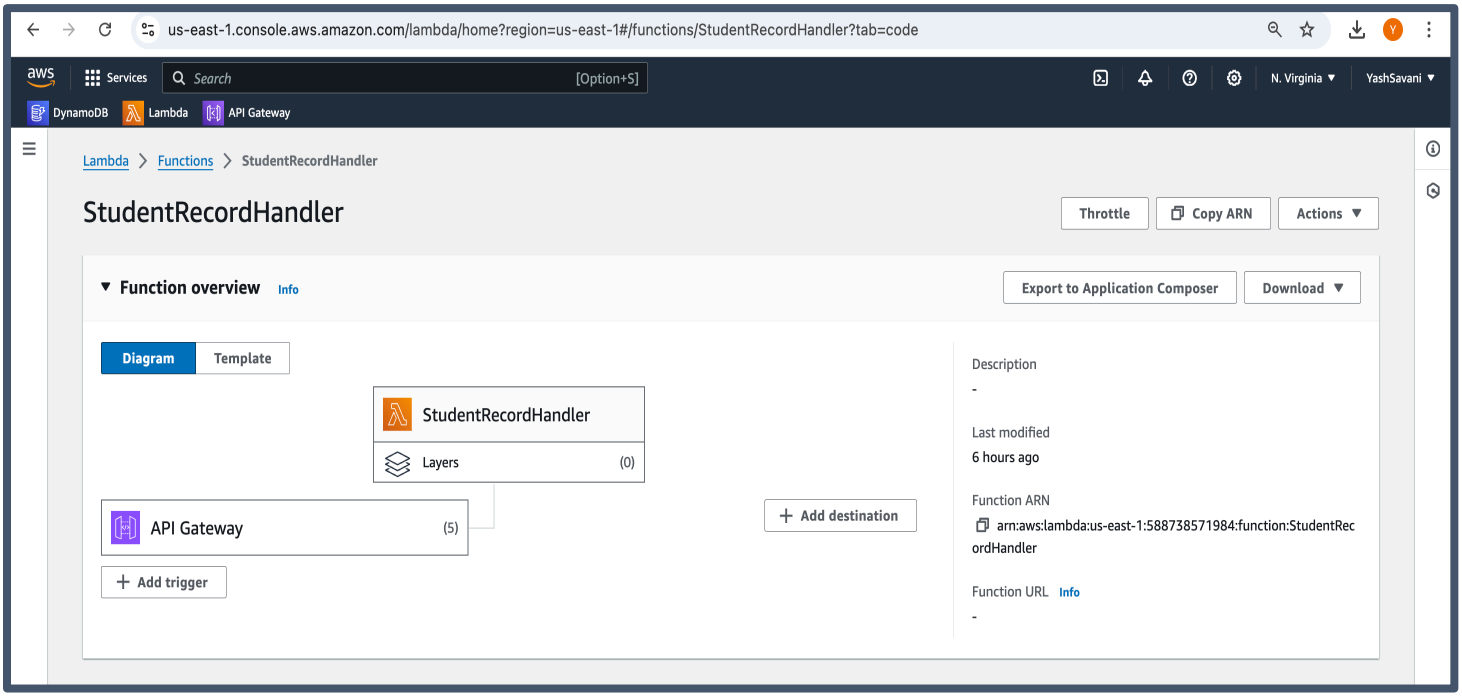


2. **Creating an AWS Lambda Function**

- Navigate to **AWS Lambda** in the AWS Management Console.
- Create a new Lambda function:
    - **Function Name**: StudentRecordHandler
    - **Runtime**: Choose Python 3.x or Node.js (depending on your preferred language).
    - **Permissions**: Attach the appropriate role to allow Lambda to read/write to DynamoDB.

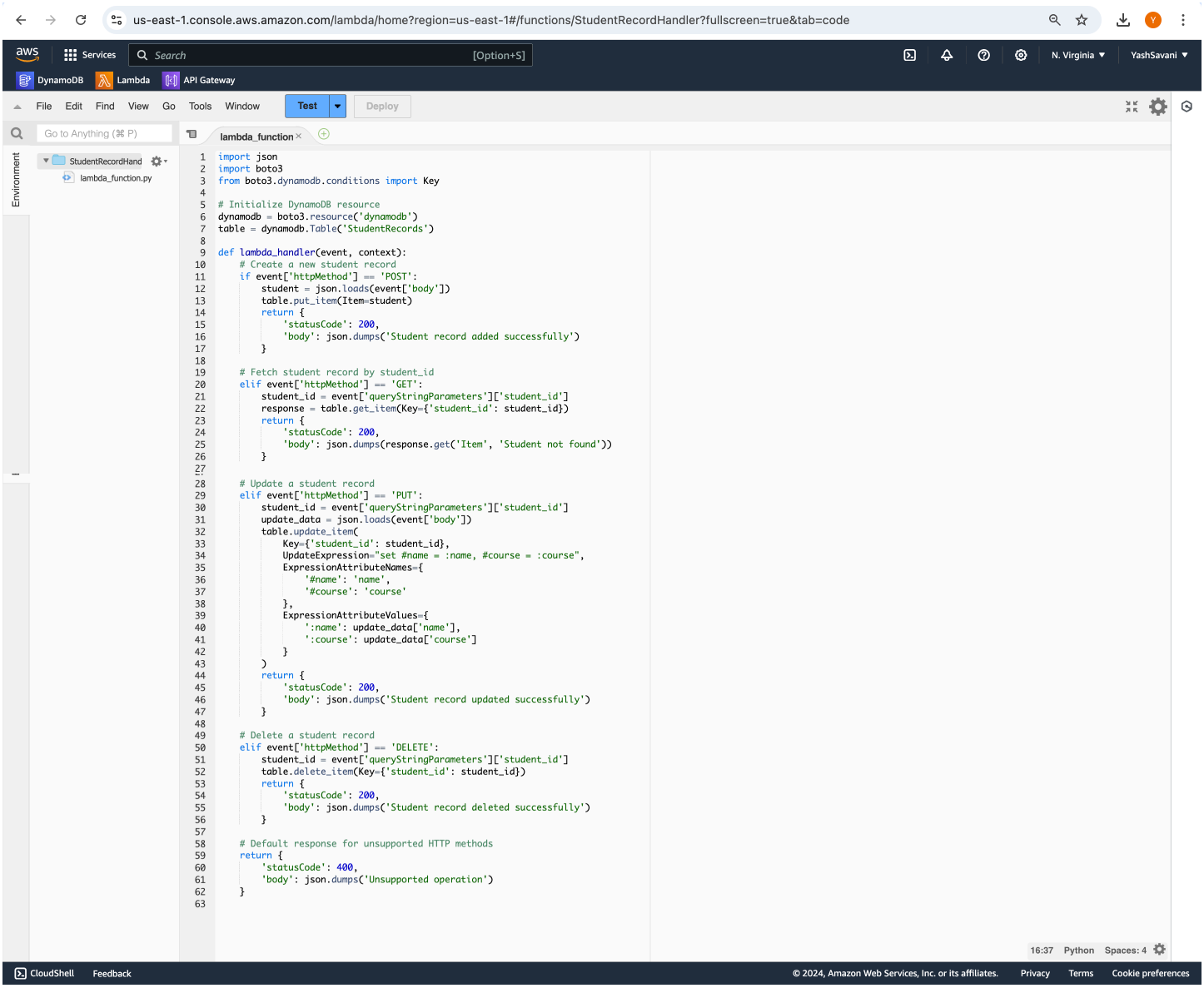






- Inside your Lambda function, write code to handle basic CRUD operations with DynamoDB.
    - **Create**: Insert a new student record into the DynamoDB table.
    - **Read**: Fetch a student record by student_id.
    - **Update**: Update a student's details.
    - **Delete**: Remove a student record.

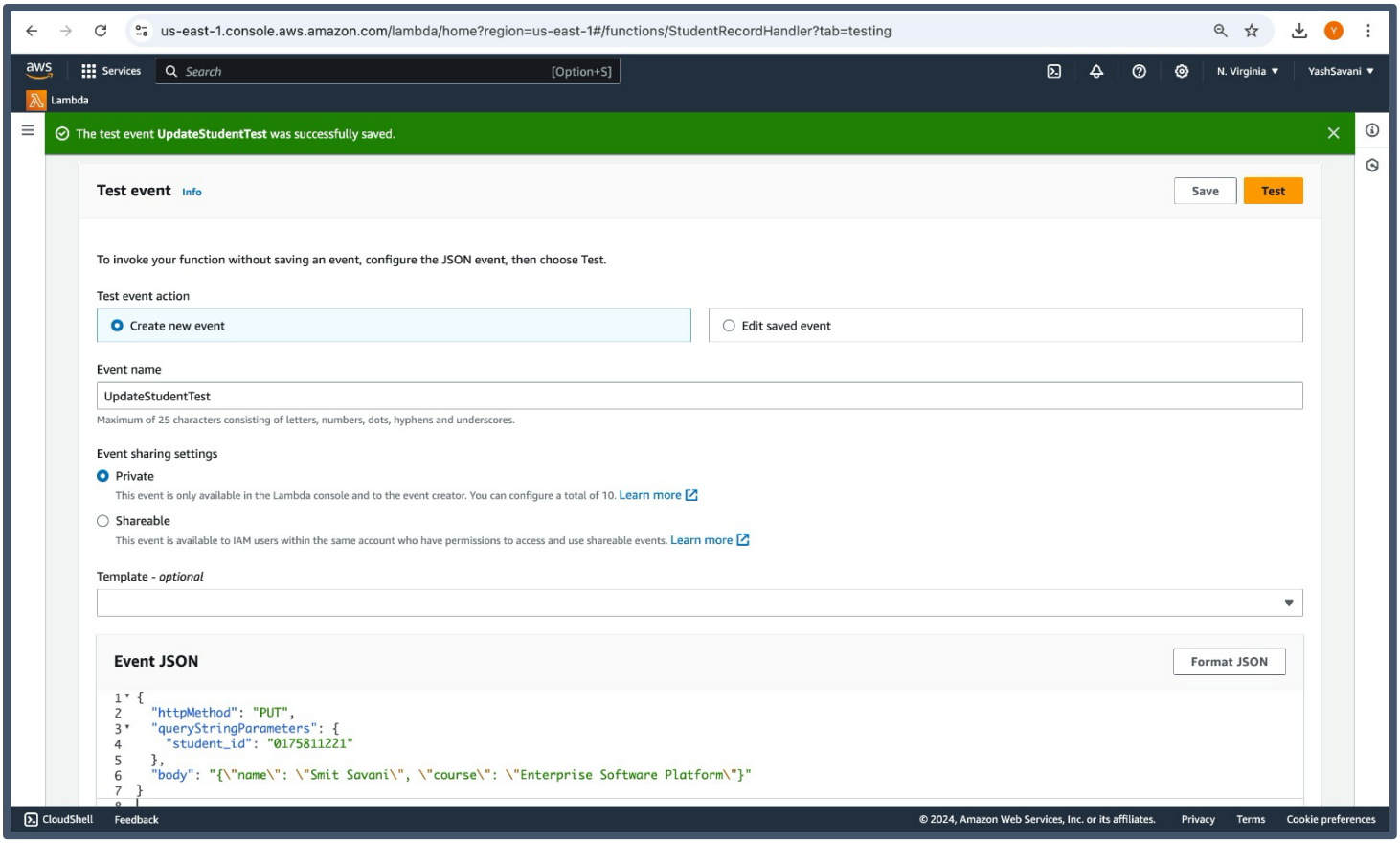
Python code for CRUD (Create, Read, Update, Delete) is given below.

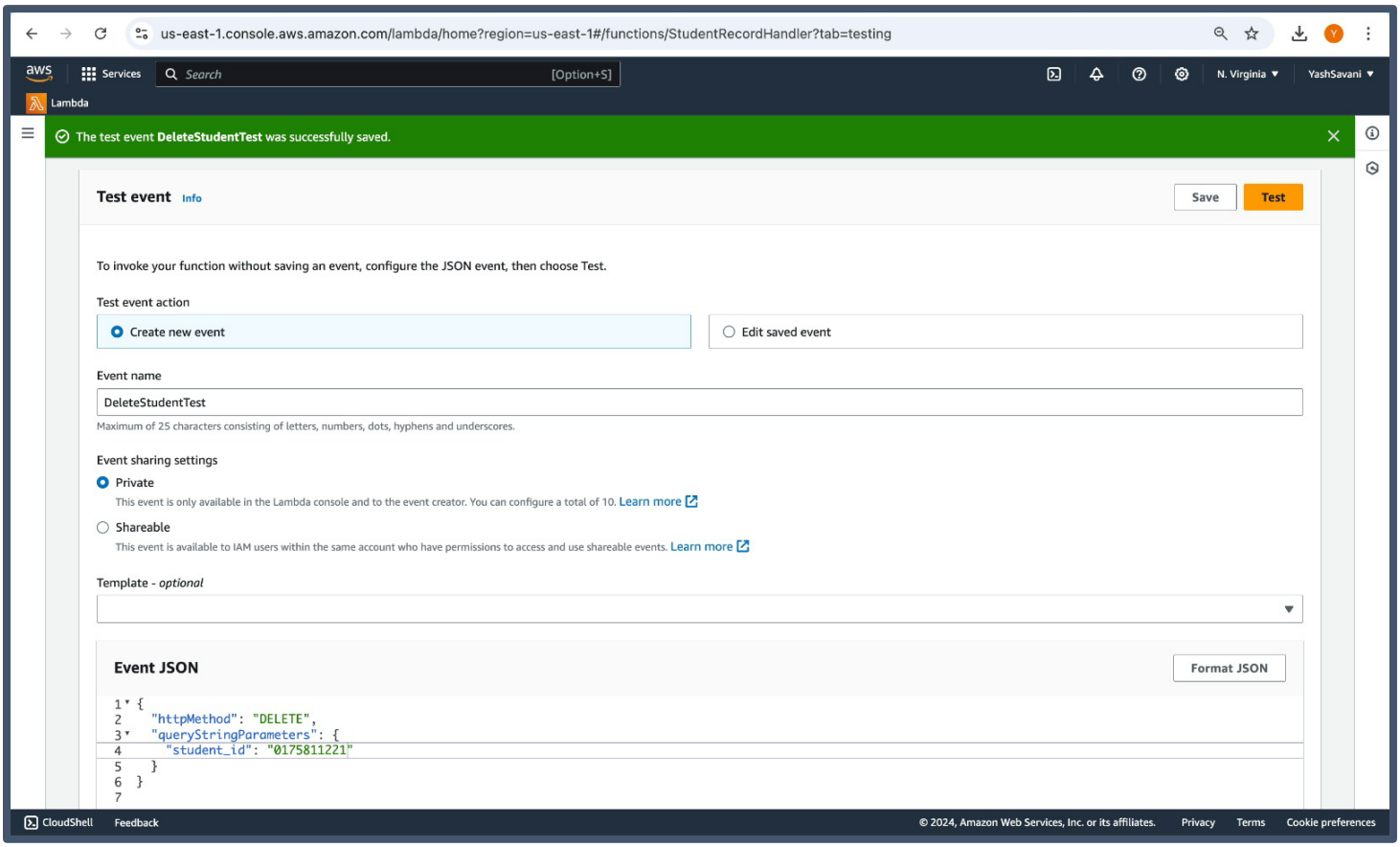


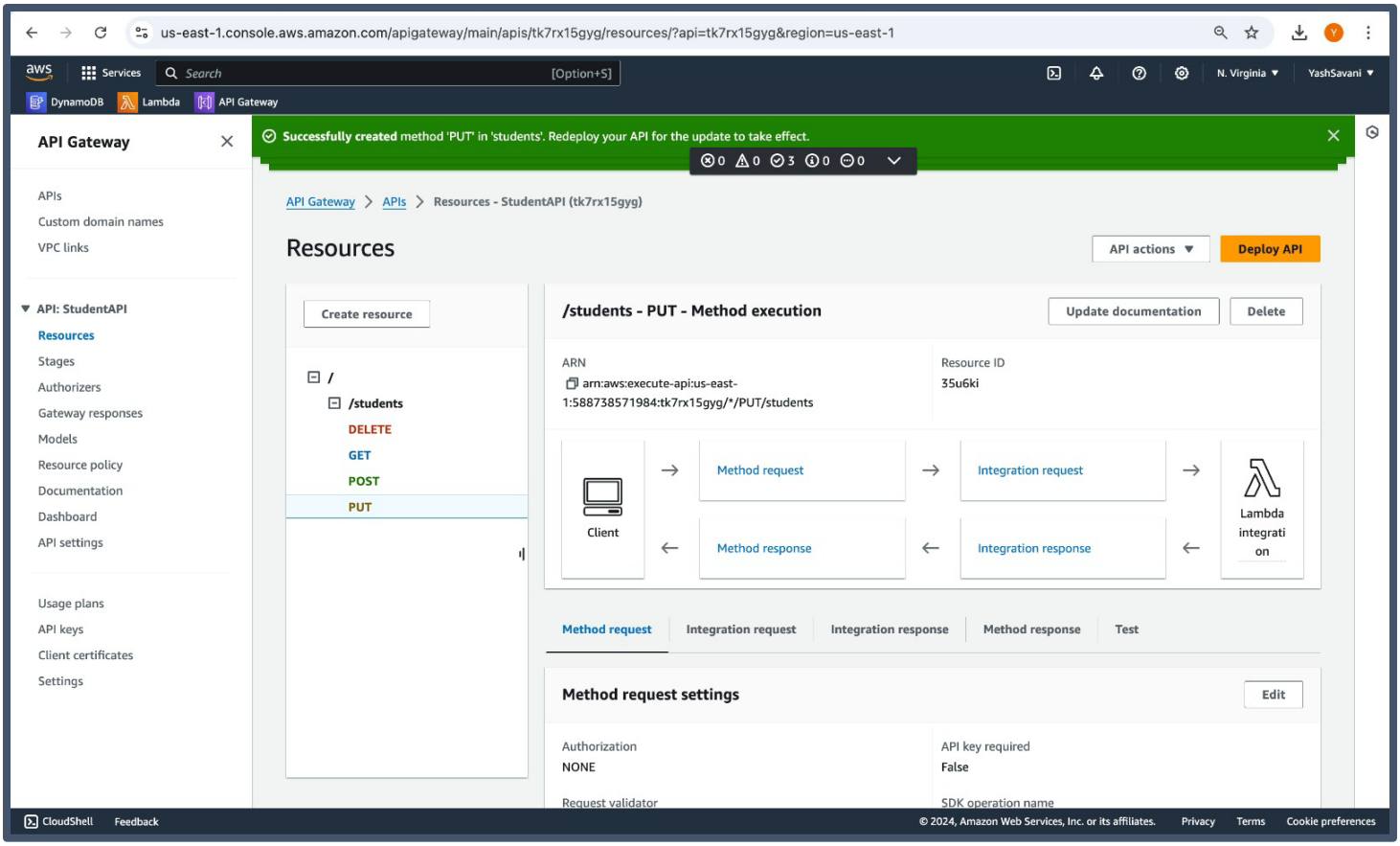
```python
import json
import boto3
from boto3.dynamodb.conditions import Key

# Initialize DynamoDB resource
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('StudentRecords')

def lambda_handler(event, context):
    # Create a new student record
    if event['httpMethod'] == 'POST':
        student = json.loads(event['body'])
        table.put_item(Item=student)
        return {
            'statusCode': 200,
            'body': json.dumps('Student record added successfully')
        }

    # Fetch student record by student_id
    elif event['httpMethod'] == 'GET':
        student_id = event['queryStringParameters']['student_id']
        response = table.get_item(Key={'student_id': student_id})
        return {
            'statusCode': 200,
            'body': json.dumps(response.get('Item', 'Student not found'))
        }

    # Update a student record
    elif event['httpMethod'] == 'PUT':
        student_id = event['queryStringParameters']['student_id']
        update_data = json.loads(event['body'])
        table.update_item(
            Key={'student_id': student_id},
            UpdateExpression="set #name = :name, #course = :course",
            ExpressionAttributeNames={
                '#name': 'name',
                '#course': 'course'
            },
            ExpressionAttributeValues={
                ':name': update_data['name'],
                ':course': update_data['course']
            }
        )
        return {
            'statusCode': 200,
            'body': json.dumps('Student record updated successfully')
        }

    # Delete a student record
    elif event['httpMethod'] == 'DELETE':
        student_id = event['queryStringParameters']['student_id']
        table.delete_item(Key={'student_id': student_id})
        return {
            'statusCode': 200,
            'body': json.dumps('Student record deleted successfully')
        }

    # Default response for unsupported HTTP methods
    return {
        'statusCode': 400,
        'body': json.dumps('Unsupported operation')
    }
```

3. Creating an API Gateway

- Go to API Gateway in the AWS Management Console.
- Create a new REST API:
    - API Name: StudentAPI
- Set up the following resources and methods:
    - POST /students: Trigger the Lambda function to add a new student.
    - GET /students: Trigger the Lambda function to retrieve student details by student_id.
    - UPDATE/students: Trigger the Lamda function to update student details by student_id.
    - DELETE/students: Trigger the Lamda function to delete student details by student_id.
- Deploy the API and note down the Invoke URL.


us-east-1.console.aws.amazon.com/lambda/home?region=us-east-1#/functions/StudentRecordHandler?tab=testing
Services
Search
[Option+S]
N. Virginia
YashSavani
Lambda
The test event UpdateStudentTest was successfully saved.
Test event Info
Save
Test
To invoke your function without saving an event, configure the JSON event, then choose Test.
Test event action
Create new event
Edit saved event
Event name
UpdateStudentTest
Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.
Event sharing settings
Private
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more
Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more
Template - optional
Event JSON
Format JSON
{
"httpMethod": "PUT",
"queryStringParameters": {
"student_id": "0175811221"
},
"body": "{\"name\": \"Smit Savani\", \"course\": \"Enterprise Software Platform\"}"
}
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/lambda/home?region=us-east-1#/functions/StudentRecordHandler?tab=testing
Services
Search
[Option+S]
N. Virginia
YashSavani
Lambda
The test event DeleteStudentTest was successfully saved.
Test event Info
Save
Test
To invoke your function without saving an event, configure the JSON event, then choose Test.
Test event action
Create new event
Edit saved event
Event name
DeleteStudentTest
Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.
Event sharing settings
Private
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more
Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more
Template - optional
Event JSON
Format JSON
{
"httpMethod": "DELETE",
"queryStringParameters": {
"student_id": "0175811221"
}
}
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

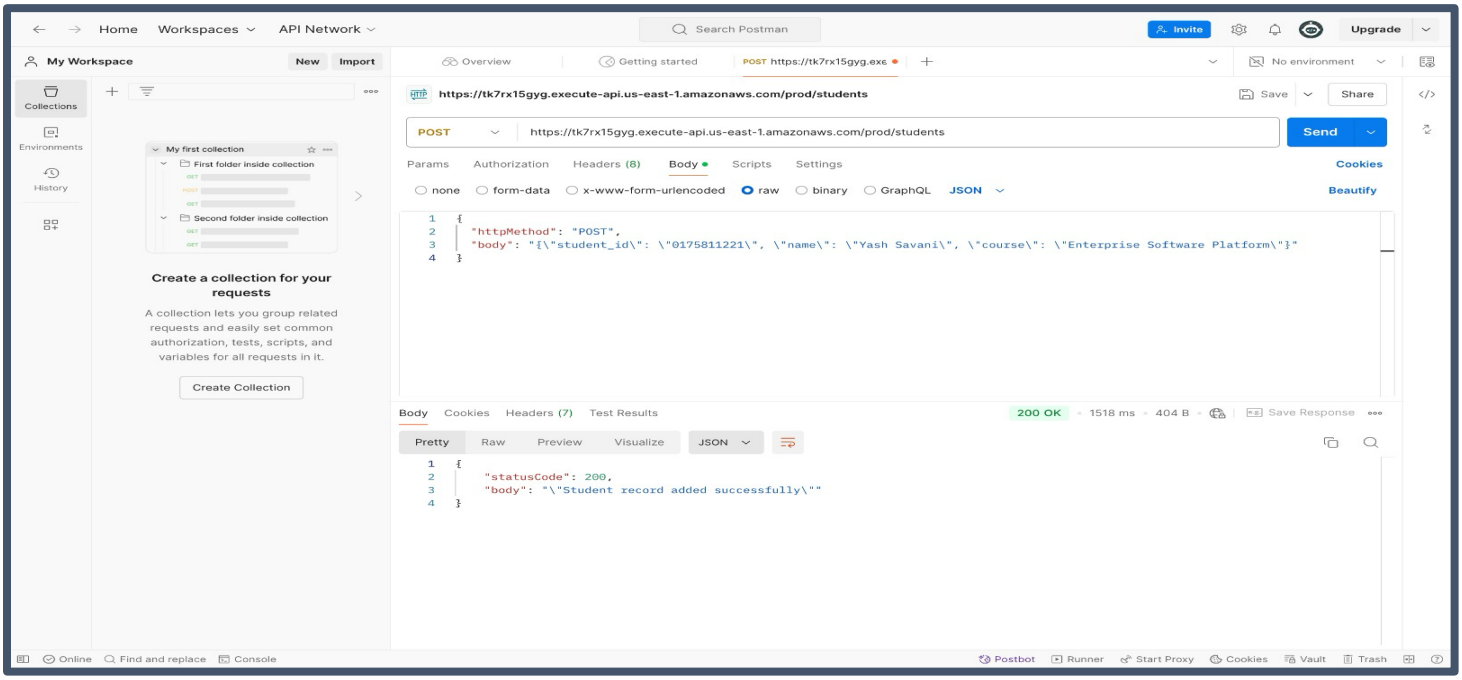


Invoke URL: https://tk7rx15gyg.execute-api.us-east-1.amazonaws.com/prod

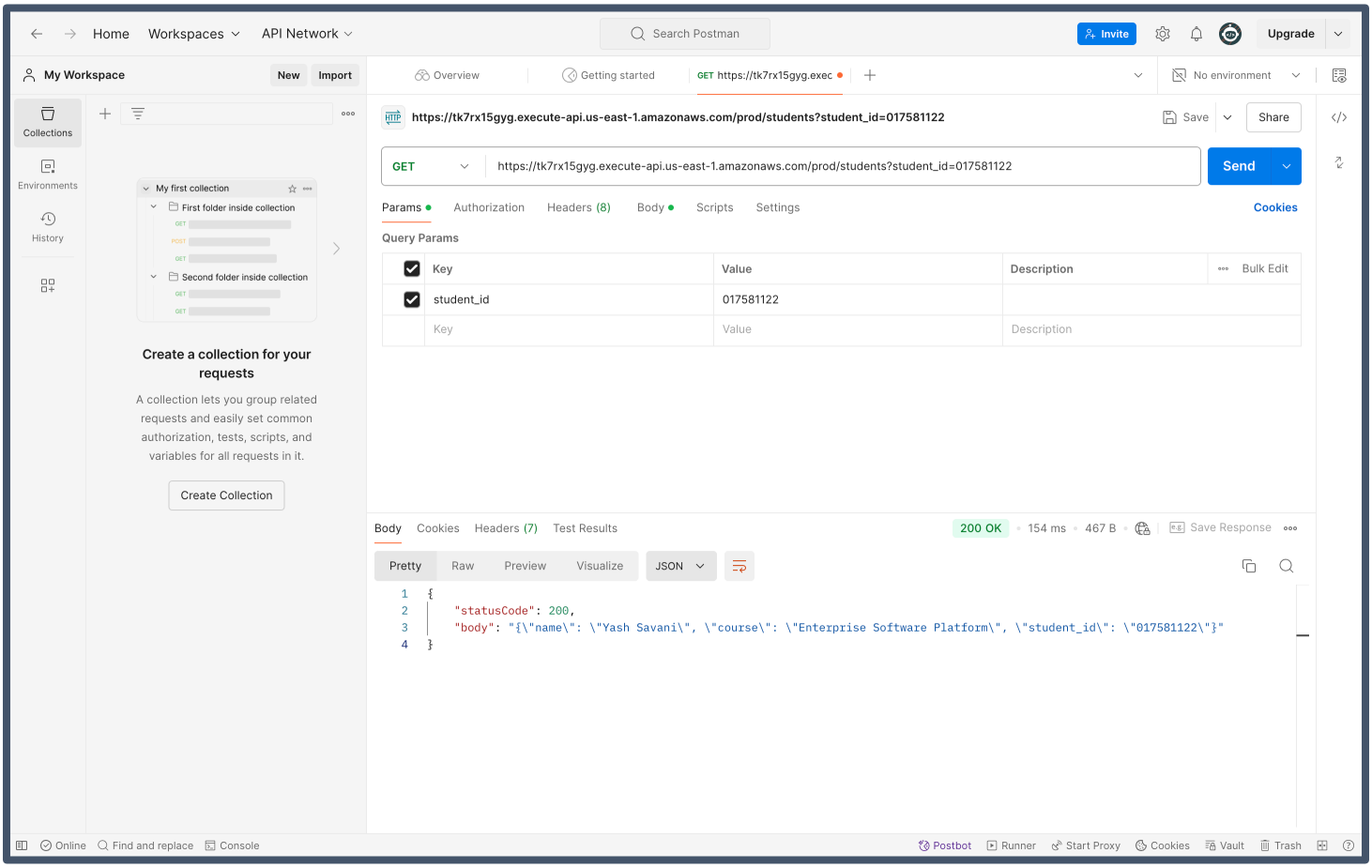
4. Testing the Application

- Use Postman or curl to test the API by sending HTTP requests to the deployed API Gateway.
- Test the following operations:
    - Create: Add a new student record.
    - Read: Retrieve the student record using the student_id.
    - Update: Update the student record using the student_id.
    - Delete: Delete the student record using the student_id.

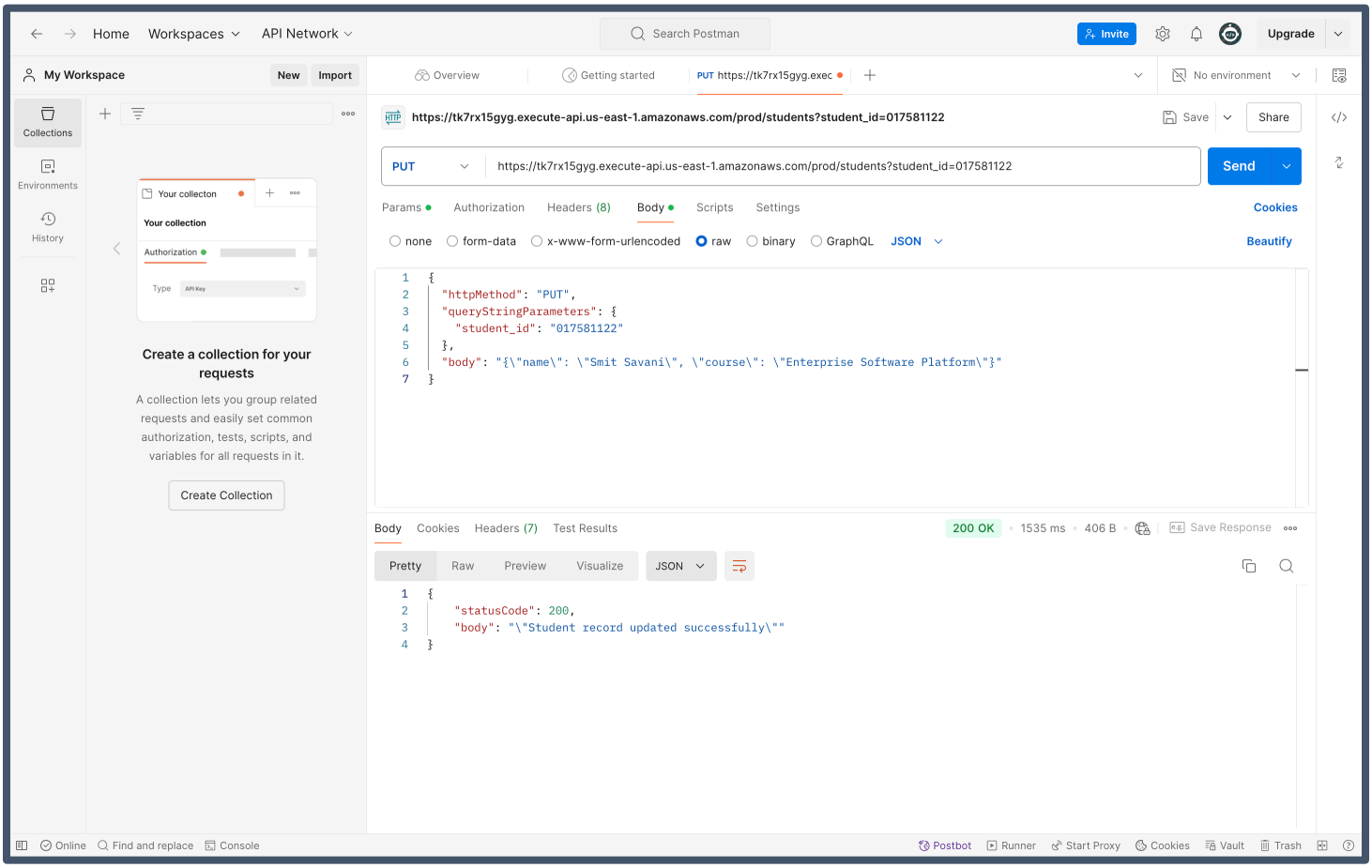
POST Method:



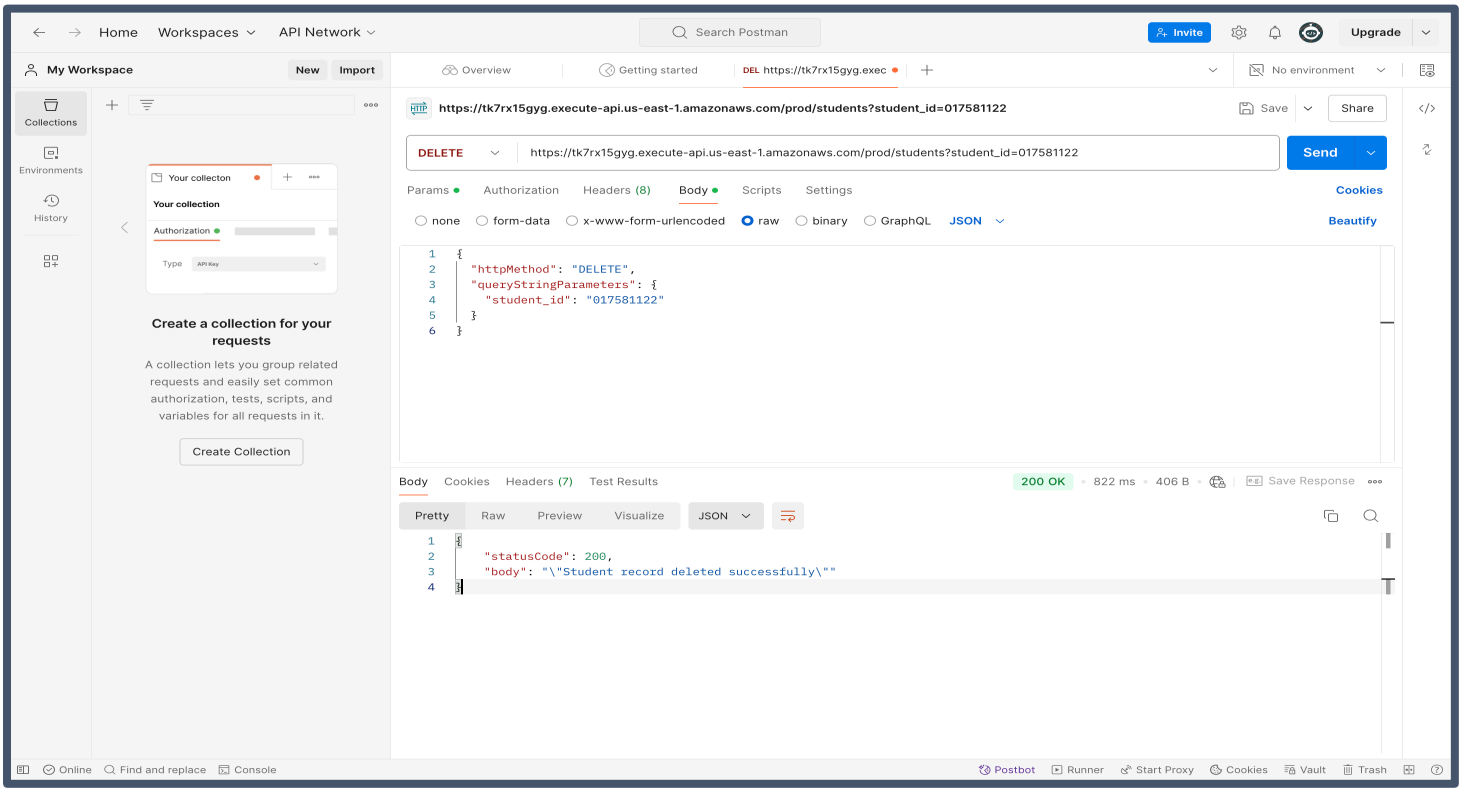
GET Method:



PUT Method:



DELETE Method:



Challenges Faced:

1. **Permission Issues**:
   Setting up the appropriate IAM permissions for the Lambda function to access DynamoDB was one of the earliest problems. **AccessDeniedException** problems were caused by improperly configured permissions, underscoring how crucial it is to comprehend AWS Identity and Access Management (IAM) policies.

2. **Event Structure Mismatches**:
   Errors relating to unexpected event structures, including missing **httpMethod** keys, were frequently encountered when testing the Lambda function. This necessitated close debugging and knowledge of the event object format used by API Gateway to activate Lambda functions.

3. **API Gateway Configuration**:
   It was a little complicated to properly set up API Gateway, including defining resources and methods and launching the API. It took careful attention to detail to configure the integration between Lambda and API Gateway, especially to make sure the right method types (POST, GET, PUT, and DELETE) were used.

4. **Error Handling and Debugging**:
   It was difficult to debug faults in a serverless setup as AWS CloudWatch logs were the only source of information. This increased intricacy in comparison to conventional debugging tools seen in regional development environments.

What I Learned:

1. **Serverless Architecture Fundamentals**:
   I acquired practical knowledge of the fundamentals of serverless computing, such as how AWS Lambda features behave as event-driven services and how well they interface with other AWS services, such as DynamoDB.
2. **Working with API Gateway**:
   I gained knowledge about how API Gateway acts as an interface for RESTful APIs that enable serverless functionality. My understanding of endpoint configuration, CORS management, and API deployment has improved as a result of this experience.
3. **Understanding IAM Roles and Policies**:
   Configuring permissions between AWS services deepened my understanding of IAM roles and policies, which are critical for maintaining the security and functionality of cloud applications.
4. **Implementing CRUD Operations in the Cloud**:
   Performing fundamental CRUD operations with DynamoDB provide insight into AWS NoSQL database management, highlighting the distinctions between cloud-native databases and conventional relational databases.
5. **Troubleshooting in the Cloud**:
   I discovered the value of using CloudWatch to monitor and debug serverless apps, which gave me a practical understanding of how to maintain and troubleshoot cloud apps in production.

Overall, through this project, I gained hands-on experience in developing and implementing a cloud-native application, showcasing the possibilities of serverless architectures and emphasizing the value of careful configuration and testing in cloud environments.