SJSU SAN JOSÉ STATE UNIVERSITY

CMPE-272
Enterprise Software Platform

Assignment: Building serverless application

SJSU ID: 017992624
Kenil Gopani
kenilghanashyambhai.gopani@sjsu.edu

Building a Serverless Web Application with AWS Lambda and DynamoDB

Steps:

Setting Up the DynamoDB Table

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

Creating an AWS Lambda Function

1. Navigate to **AWS Lambda** in the AWS Management Console.
2. 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.
3. 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.

Creating an API Gateway

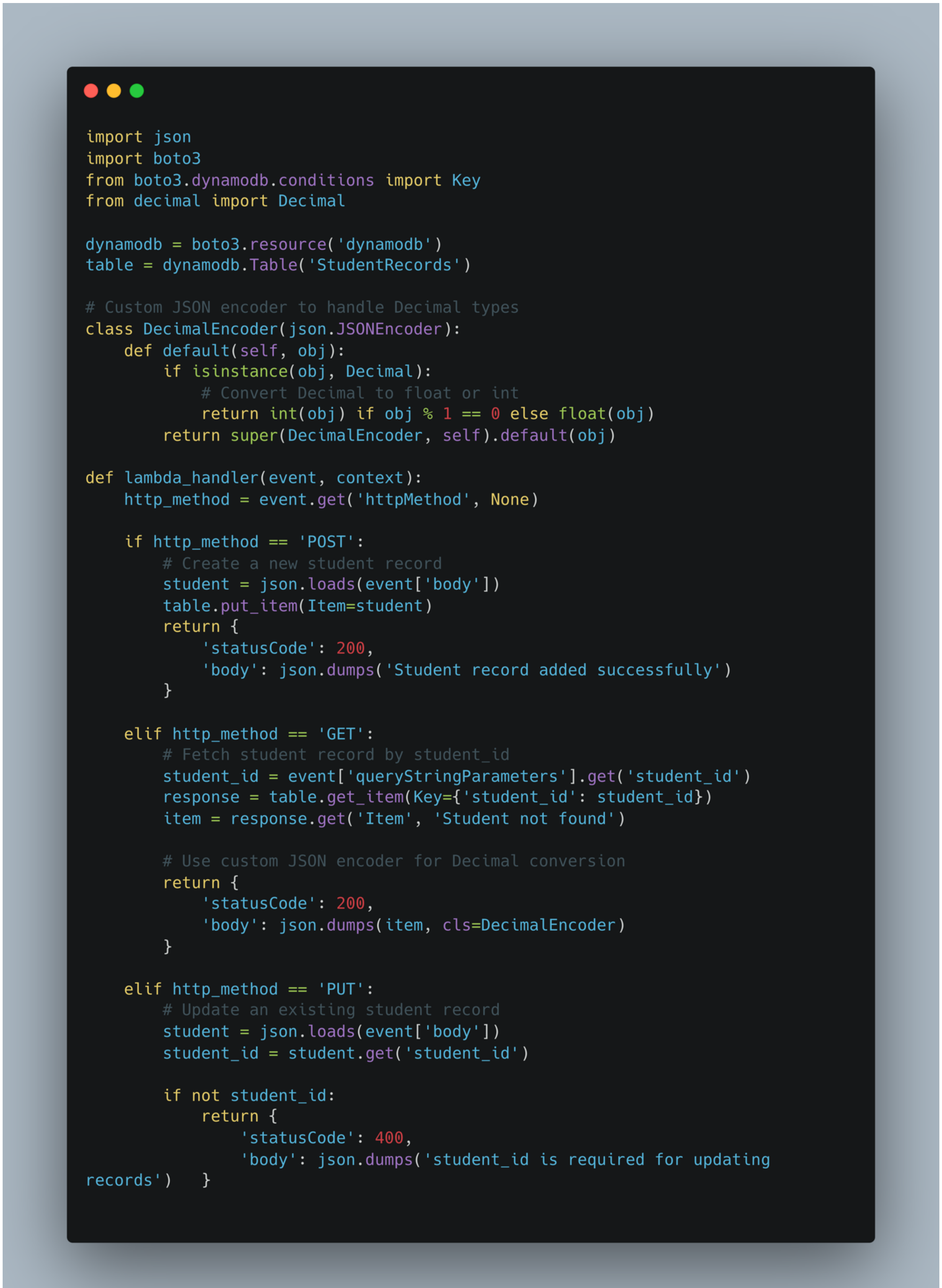
1. Go to **API Gateway** in the AWS Management Console.
2. Create a new REST API:
   - **API Name**: `StudentAPI`
3. Set up the following resources and methods:
   - **POST /students**: Trigger the Lambda function to add a new student.
   - **GET /students**: Trigger the Lambda function to retrieve student details by `student_id`.
4. Deploy the API and note down the **Invoke URL**.

Testing the Application

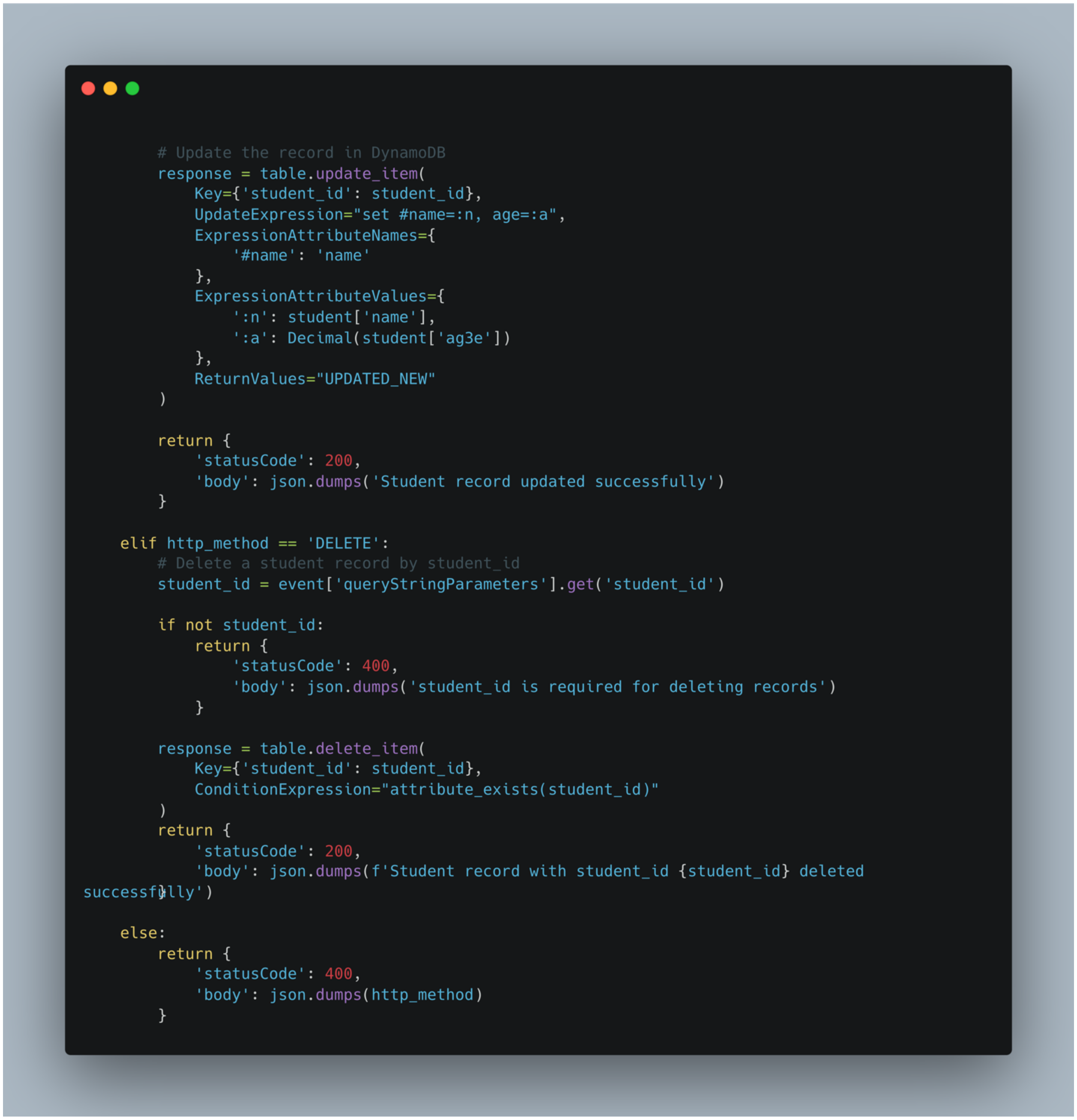
1. Use **Postman** or **curl** to test the API by sending HTTP requests to the deployed API Gateway.

2. Test the following operations:
    - **Create**: Add a new student record.
    - **Read**: Retrieve the student record using the `student_id`.

Code: Lambda Function for CRUD Operations:



```
import json
import boto3
from boto3.dynamodb.conditions import Key
from decimal import Decimal

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

# Custom JSON encoder to handle Decimal types
class DecimalEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Decimal):
            # Convert Decimal to float or int
            return int(obj) if obj % 1 == 0 else float(obj)
        return super(DecimalEncoder, self).default(obj)

def lambda_handler(event, context):
    http_method = event.get('httpMethod', None)

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

    elif http_method == 'GET':
        # Fetch student record by student_id
        student_id = event['queryStringParameters'].get('student_id')
        response = table.get_item(Key={'student_id': student_id})
        item = response.get('Item', 'Student not found')

        # Use custom JSON encoder for Decimal conversion
        return {
            'statusCode': 200,
            'body': json.dumps(item, cls=DecimalEncoder)
        }

    elif http_method == 'PUT':
        # Update an existing student record
        student = json.loads(event['body'])
        student_id = student.get('student_id')

        if not student_id:
            return {
                'statusCode': 400,
                'body': json.dumps('student_id is required for updating
records')   }
```

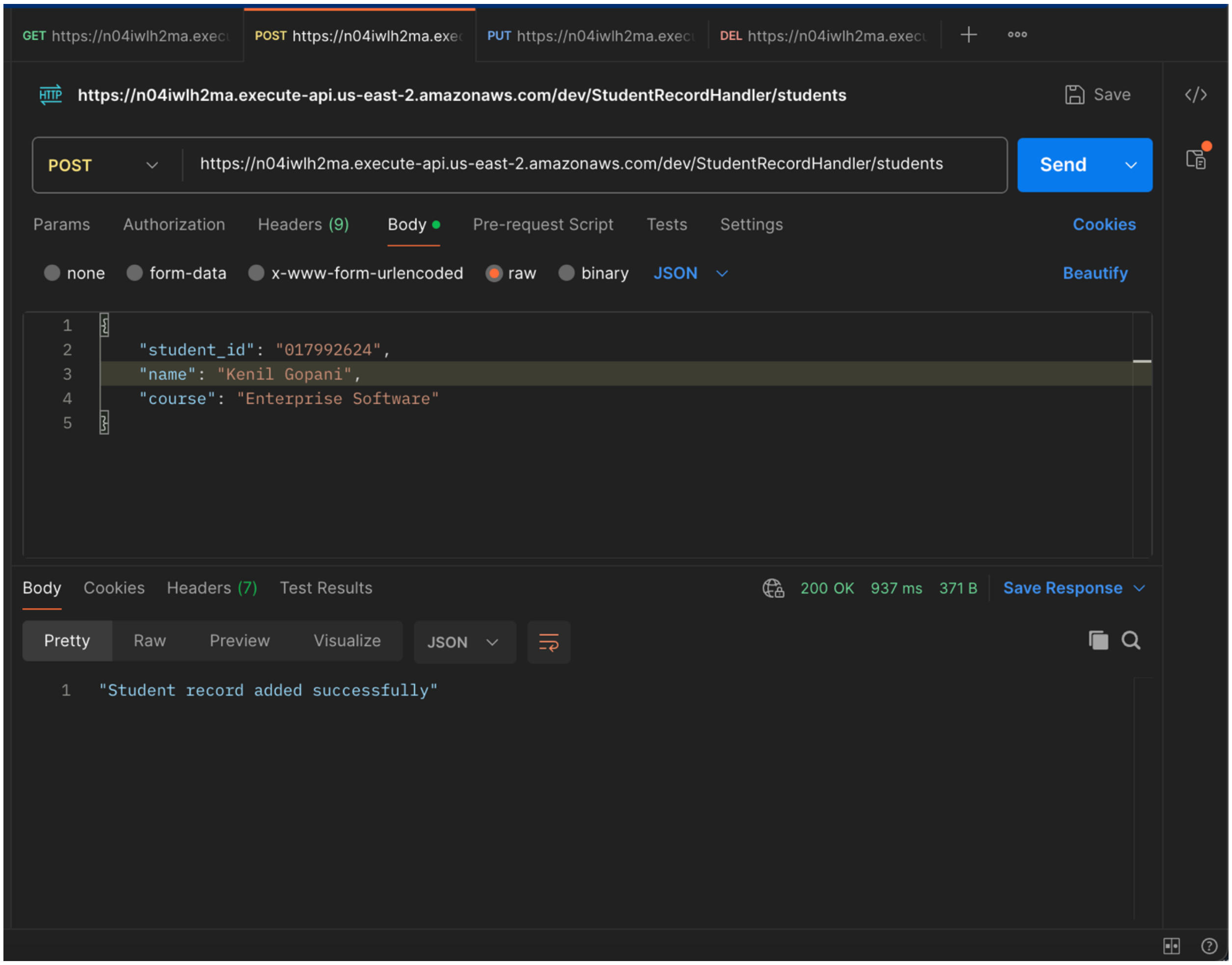


```
        # Update the record in DynamoDB
        response = table.update_item(
            Key={'student_id': student_id},
            UpdateExpression="set #name=:n, age=:a",
            ExpressionAttributeNames={
                '#name': 'name'
            },
            ExpressionAttributeValues={
                ':n': student['name'],
                ':a': Decimal(student['ag3e'])
            },
            ReturnValues="UPDATED_NEW"
        )

        return {
            'statusCode': 200,
            'body': json.dumps('Student record updated successfully')
        }

    elif http_method == 'DELETE':
        # Delete a student record by student_id
        student_id = event['queryStringParameters'].get('student_id')

        if not student_id:
            return {
                'statusCode': 400,
                'body': json.dumps('student_id is required for deleting records')
            }

        response = table.delete_item(
            Key={'student_id': student_id},
            ConditionExpression="attribute_exists(student_id)"
        )
        return {
            'statusCode': 200,
            'body': json.dumps(f'Student record with student_id {student_id} deleted
successfully')
        }

    else:
        return {
            'statusCode': 400,
            'body': json.dumps(http_method)
        }
```

Screenshots:

APIs

1. POST Request: https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students



https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students

POST https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students

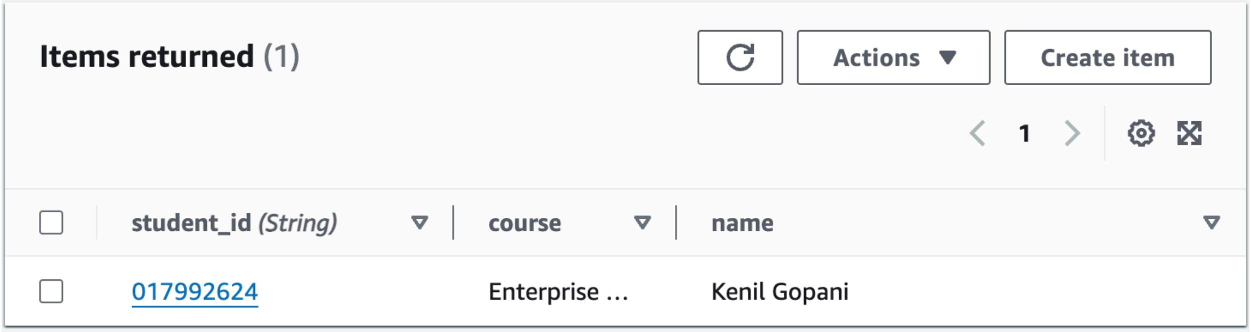
```
{
    "student_id": "017992624",
    "name": "Kenil Gopani",
    "course": "Enterprise Software"
}
```

200 OK 937 ms 371 B

JSON

```
"Student record added successfully"
```

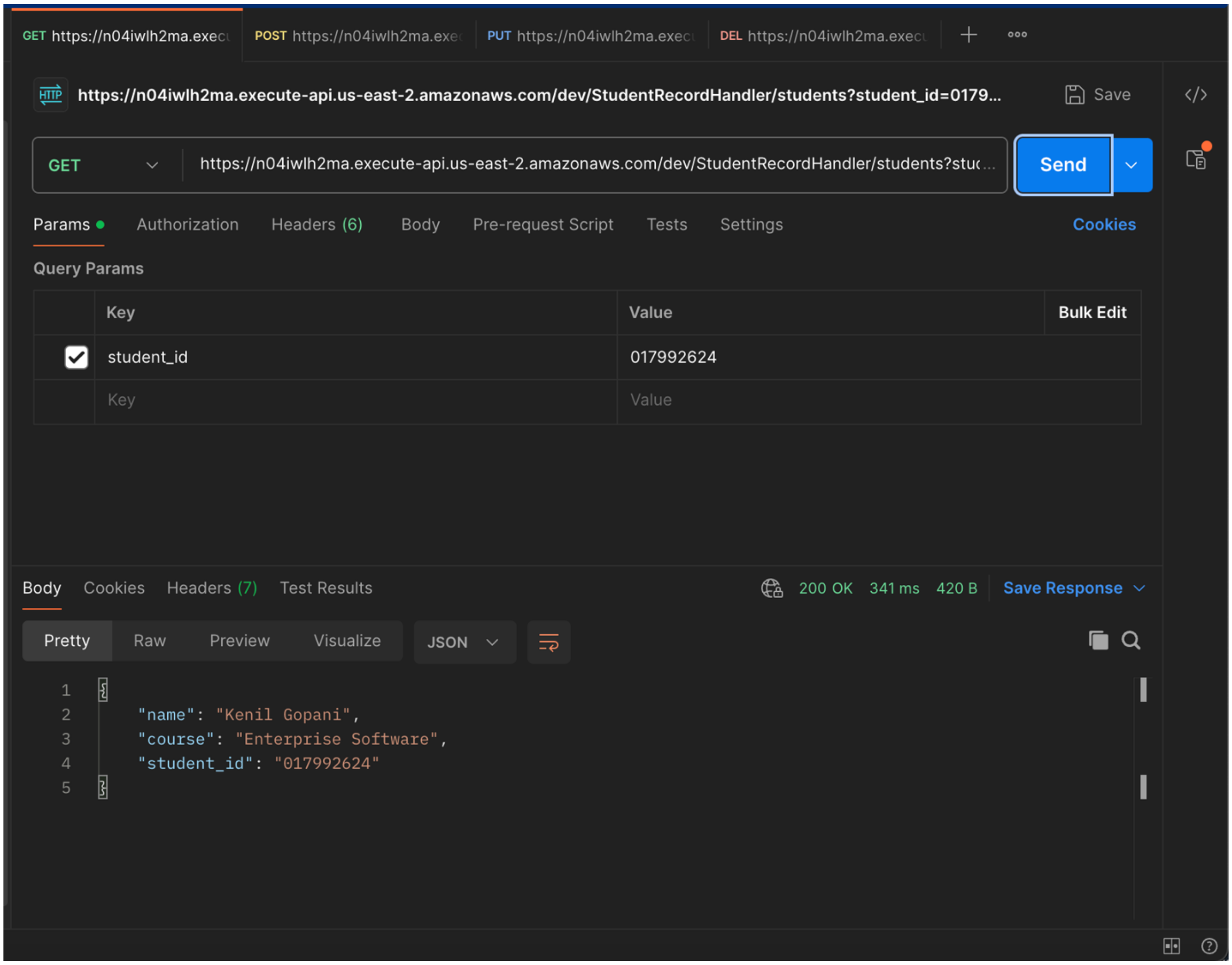
- Database



Items returned (1)

□	student_id *(String)*	course	name
□	017992624	Enterprise …	Kenil Gopani

2. GET Request: https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?student_id=017992624



https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?student_id=0179...

GET https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?stuc...

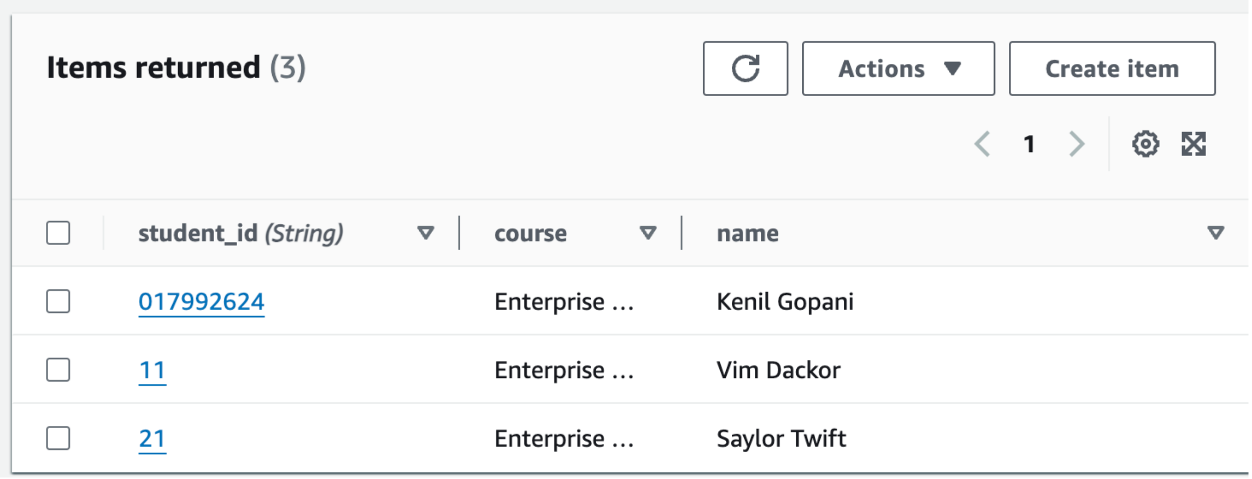
Query Params

	Key	Value	Bulk Edit
☑	student_id	017992624	
	Key	Value	

200 OK 341 ms 420 B

```
{
    "name": "Kenil Gopani",
    "course": "Enterprise Software",
    "student_id": "017992624"
}
```

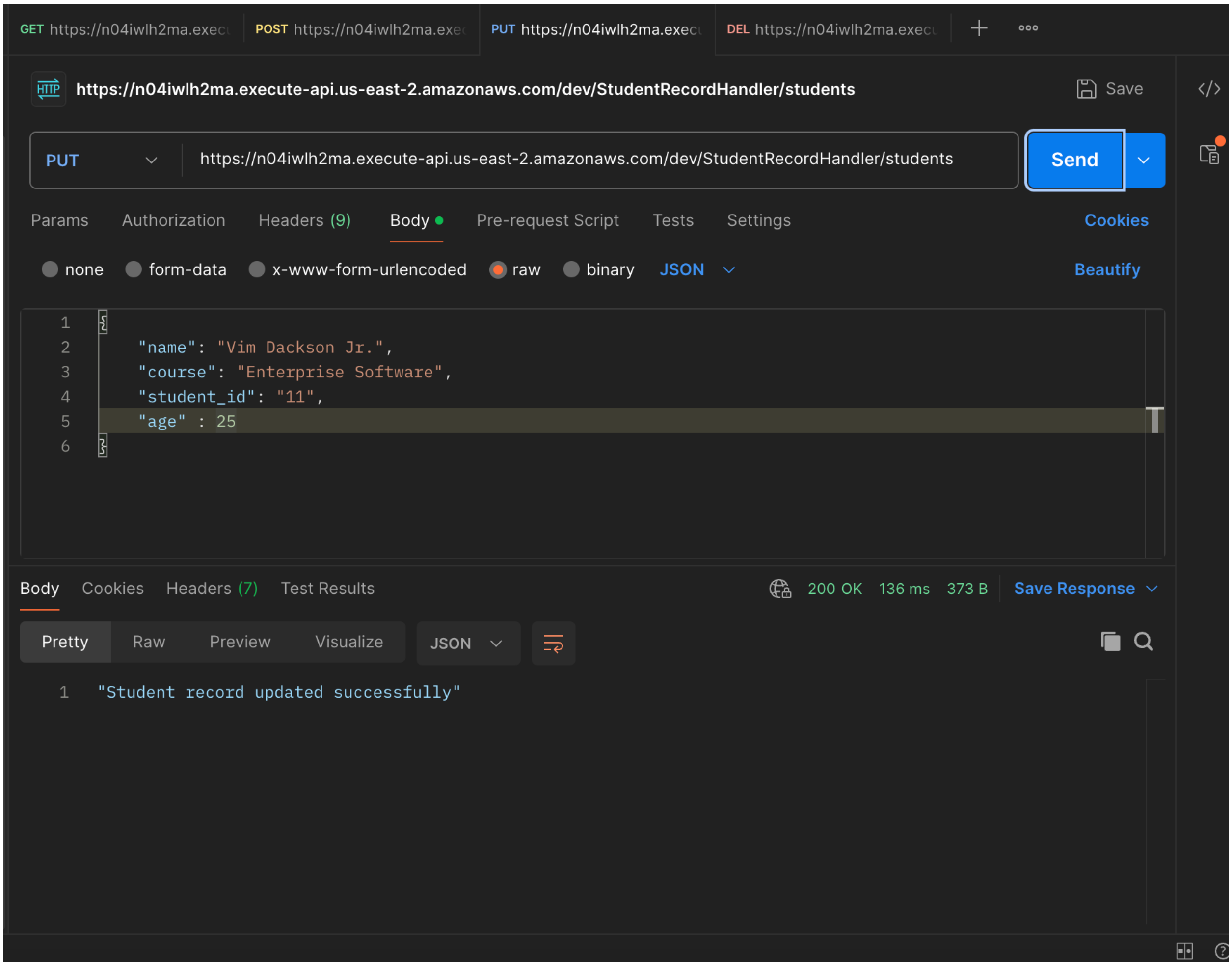
Let’s add some more records:



Items returned (3)

□	student_id *(String)*	course	name
□	017992624	Enterprise ...	Kenil Gopani
□	11	Enterprise ...	Vim Dackor
□	21	Enterprise ...	Saylor Twift

3. PUT Request: https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students



https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students

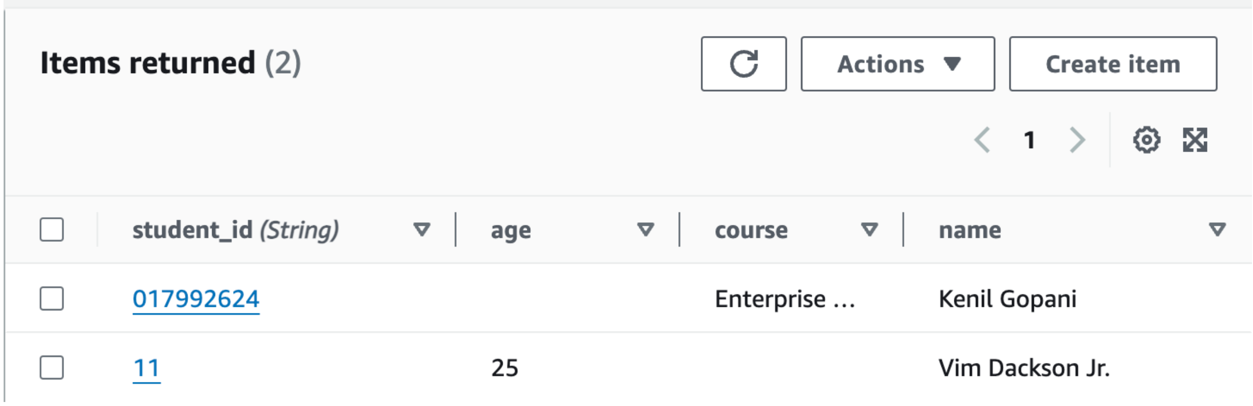
PUT https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students

```json
{
    "name": "Vim Dackson Jr.",
    "course": "Enterprise Software",
    "student_id": "11",
    "age" : 25
}
```

200 OK 136 ms 373 B

```
"Student record updated successfully"
```

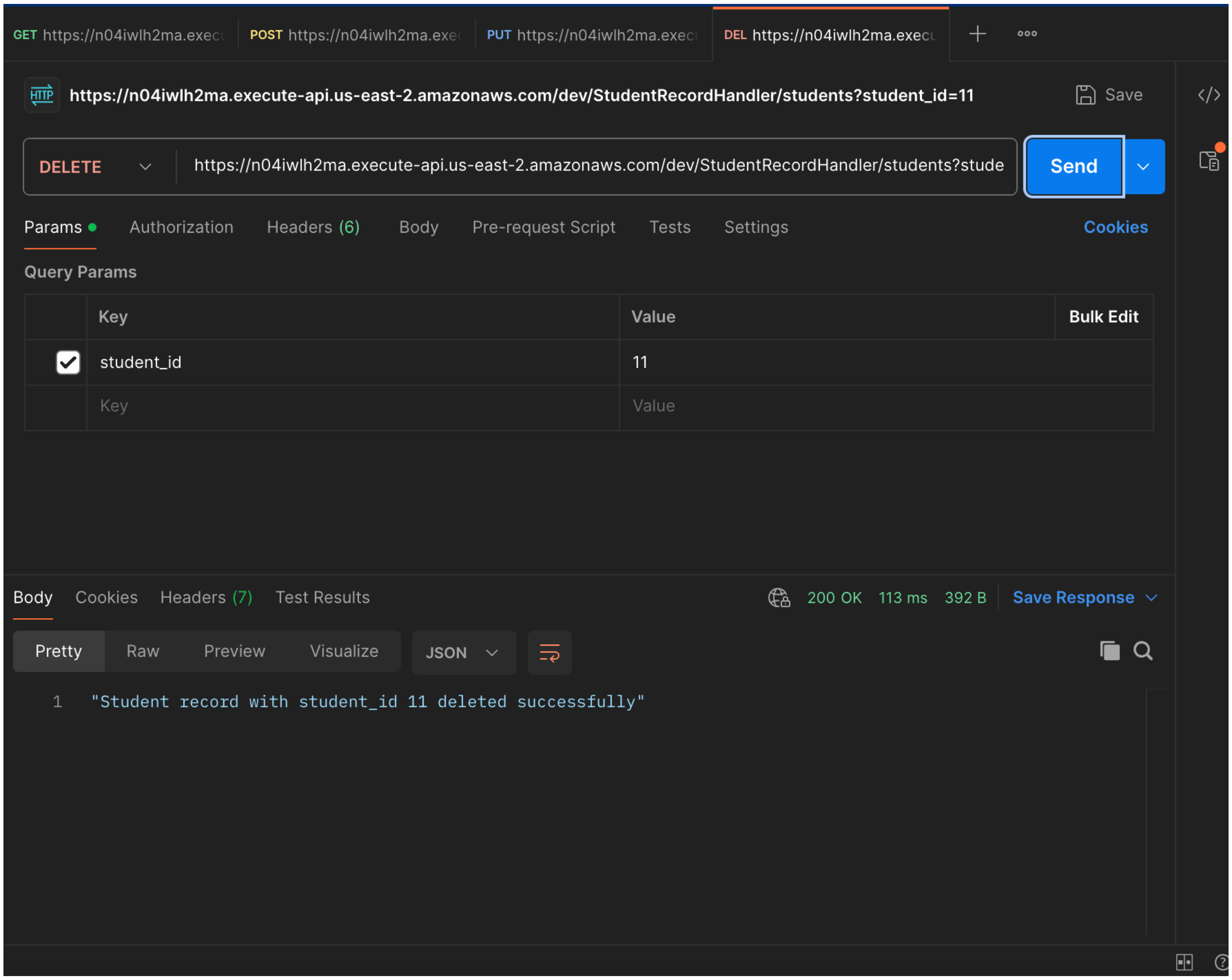
- Database



Items returned (2)

□	student_id *(String)*	age	course	name
□	017992624		Enterprise …	Kenil Gopani
□	11	25		Vim Dackson Jr.

4. DELETE Request: https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?student_id=21



https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?student_id=11

DELETE https://n04iwlh2ma.execute-api.us-east-2.amazonaws.com/dev/StudentRecordHandler/students?stude

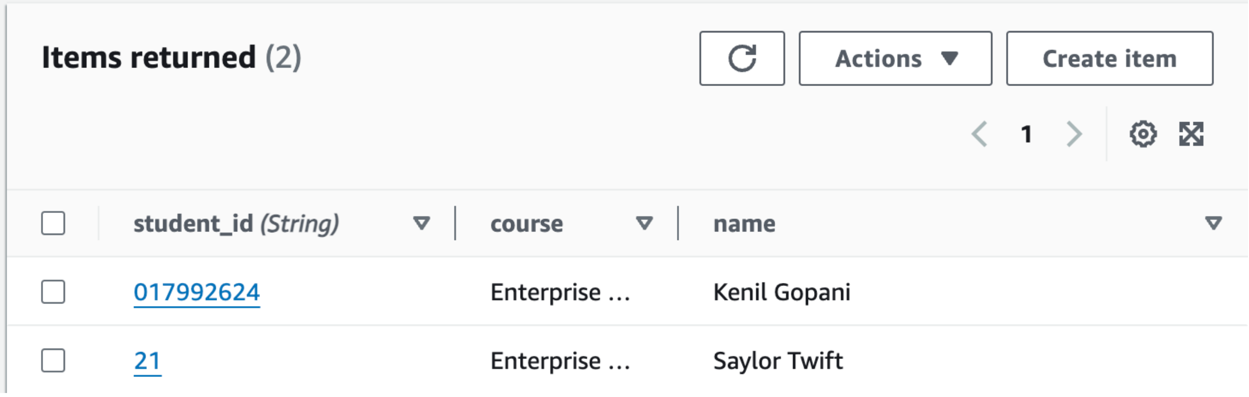
Query Params

	Key	Value	Bulk Edit
☑	student_id	11	
	Key	Value	

200 OK 113 ms 392 B

```
"Student record with student_id 11 deleted successfully"
```

- Database



Items returned (2)

□	student_id *(String)*	course	name
□	017992624	Enterprise …	Kenil Gopani
□	21	Enterprise …	Saylor Twift

Reflection :

- Working with AWS Lambda provided me with insights into how it differs significantly from traditional services like AWS EC2. Unlike EC2, which requires manual scaling and continuous operation, Lambda is auto-scalable and event-driven. It executes code only when invoked and then returns to an idle state, making it highly efficient for on-demand operations.
- I learned how Lambda can be leveraged for various backend tasks such as notification broadcasting, image compression, and more, making it a versatile tool. Its cost-effectiveness especially for tasks with intermittent workloads. However, I also recognized some limitations, such as cold start delays and a limited execution time for long-running tasks, which need to be considered depending on the use case.
- Another interesting aspect was DynamoDB's integration with AWS Lambda. This integration made it easier to build a fully serverless application, allowing me to quickly read and write data without managing any infrastructure.