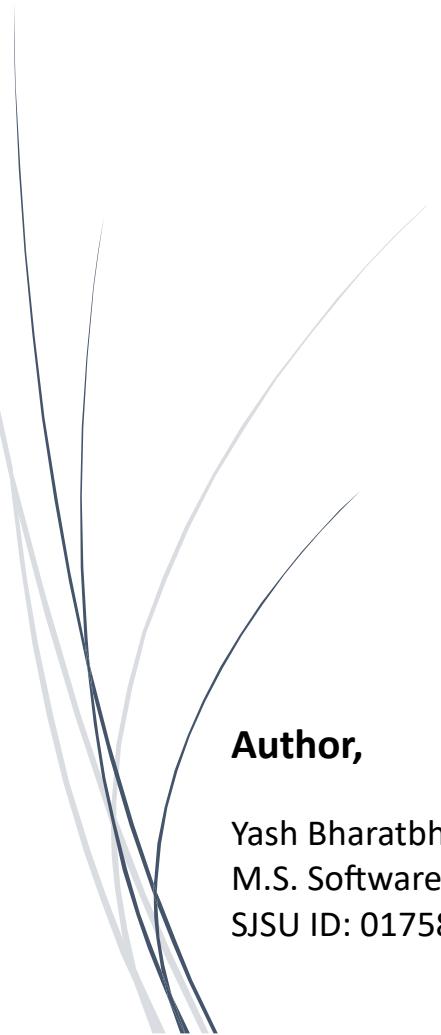




## Assignment 2

# Building serverless application

[CMPE272 - Enterprise Software Platform]



A series of thin, light gray curved lines are drawn from the bottom left towards the center of the page, creating a dynamic, flowing effect.

**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.

The screenshot shows the 'Create table' wizard in the AWS DynamoDB console. The URL in the browser is [us-east-1.console.aws.amazon.com/dynamodbv2/home?region=us-east-1#create-table](https://us-east-1.console.aws.amazon.com/dynamodbv2/home?region=us-east-1#create-table). The navigation bar includes 'Services' and 'Search' tabs, and the region is set to 'N. Virginia'. The main page shows the 'Create table' step, with the 'Table details' section filled out. The table name is 'StudentRecords' and the partition key is 'student\_id' of type String. Below this, there is a 'Sort key - optional' field with an empty input. The 'Table settings' section at the bottom has a radio button selected for 'Default settings'. At the bottom of the page are links for 'CloudShell', 'Feedback', and copyright information: '© 2024, Amazon Web Services, Inc. or its affiliates.' and 'Privacy Terms Cookie preferences'.

The screenshot shows the AWS DynamoDB console with a green success message at the top: "The StudentRecords table was created successfully." Below this, the "Tables (1) Info" section displays a single table named "StudentRecords". The table details are as follows:

Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size	Table class
StudentRecords	Active	student_id (S)	-	0	Off	Provisioned (5)	Provisioned (5)	0 bytes	Standard

At the bottom of the page, there are links for CloudShell, Feedback, and a footer with copyright information.

## 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.

The screenshot shows the AWS Lambda console with the "Create function" wizard open. The first step, "Choose one of the following options to create your function.", has three options:

- Author from scratch**: Selected. Description: Start with a simple Hello World example.
- Use a blueprint**: Description: Build a Lambda application from sample code and configuration presets for common use cases.
- Container image**: Description: Select a container image to deploy for your function.

The "Basic information" step is currently active, showing the following fields:

- Function name**: StudentRecordHandle
- Runtime**: Python 3.12
- Architecture**: x86\_64
- Permissions**: 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.

At the bottom of the wizard, there is a link to "Change default execution role". The footer of the page includes CloudShell, Feedback, and standard AWS footer links.

The screenshot shows the AWS Lambda console interface. On the left, there's a sidebar with navigation links like Dashboard, Applications, Functions, Additional resources, and Related AWS resources. The main area is titled "Functions (1)" and shows a table with one row for "StudentRecordHandler". The table columns include Function name, Description, Package type, Runtime, and Last modified. The function details are: Function name - StudentRecordHandler, Description - -, Package type - Zip, Runtime - Python 3.12, and Last modified - 6 hours ago. There are buttons for Actions, Create function, and a search bar at the top.

The screenshot shows the "Function overview" page for the "StudentRecordHandler" function. It includes tabs for Diagram and Template. The Diagram tab displays a visual representation of the function's triggers and destinations. Triggers shown are API Gateway (5). Destinations shown are Layers (0) and another API Gateway entry. Buttons for + Add destination and + Add trigger are available. To the right, there are sections for Description (empty), Last modified (6 hours ago), Function ARN (arn:aws:lambda:us-east-1:588738571984:function:StudentRecordHandler), and Function URL (Info). Buttons for Throttle, Copy ARN, and Actions are also present.

- 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.

The screenshot shows the AWS Lambda function editor. The left sidebar lists 'Environment' and 'lambda\_function'. The main area contains the following Python code:

```
1 import json
2 import boto3
3 from boto3.dynamodb.conditions import Key
4
5 # Initialize DynamoDB resource
6 dynamodb = boto3.resource('dynamodb')
7 table = dynamodb.Table('StudentRecords')
8
9 def lambda_handler(event, context):
10     # Create a new student record
11     if event['httpMethod'] == 'POST':
12         student = json.loads(event['body'])
13         table.put_item(Item=student)
14         return {
15             'statusCode': 200,
16             'body': json.dumps('Student record added successfully')
17         }
18
19     # Fetch student record by student_id
20     elif event['httpMethod'] == 'GET':
21         student_id = event['queryStringParameters']['student_id']
22         response = table.get_item(Key={'student_id': student_id})
23         return {
24             'statusCode': 200,
25             'body': json.dumps(response.get('Item', 'Student not found'))
26         }
27
28     # Update a student record
29     elif event['httpMethod'] == 'PUT':
30         student_id = event['queryStringParameters']['student_id']
31         update_data = json.loads(event['body'])
32         table.update_item(
33             Key={'student_id': student_id},
34             UpdateExpression="set #name = :name, #course = :course",
35             ExpressionAttributeNames={
36                 '#name': 'name',
37                 '#course': 'course'
38             },
39             ExpressionAttributeValues={
40                 ':name': update_data['name'],
41                 ':course': update_data['course']
42             }
43         )
44         return {
45             'statusCode': 200,
46             'body': json.dumps('Student record updated successfully')
47         }
48
49     # Delete a student record
50     elif event['httpMethod'] == 'DELETE':
51         student_id = event['queryStringParameters']['student_id']
52         table.delete_item(Key={'student_id': student_id})
53         return {
54             'statusCode': 200,
55             'body': json.dumps('Student record deleted successfully')
56         }
57
58     # Default response for unsupported HTTP methods
59     return {
60         'statusCode': 400,
61         'body': json.dumps('Unsupported operation')
62     }
63
```

The code implements a Lambda function named 'lambda\_handler' that handles four HTTP methods: POST, GET, PUT, and DELETE. It interacts with a DynamoDB table called 'StudentRecords'. The POST method adds a new student record. The GET method retrieves a student record by its ID. The PUT method updates an existing student record. The DELETE method deletes a student record by its ID. For unsupported methods, it returns a 400 Bad Request error.

### 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 Lambda function to update student details by student\_id.
  - DELETE/students: Trigger the Lambda function to delete student details by student\_id.
- Deploy the API and note down the Invoke URL.

The test event **UpdateStudentTest** was successfully saved.

**Test event** [Info](#)

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](#)

```
1 * {  
2   "httpMethod": "PUT",  
3   "queryStringParameters": {  
4     "student_id": "0175811221"  
5   },  
6   "body": "{\"name\": \"Smit Savani\", \"course\": \"Enterprise Software Platform\"}"  
7 }
```

CloudShell Feedback © 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

The test event **DeleteStudentTest** was successfully saved.

**Test event** [Info](#)

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](#)

```
1 * {  
2   "httpMethod": "DELETE",  
3   "queryStringParameters": {  
4     "student_id": "0175811221"  
5   },  
6 }  
7 
```

CloudShell Feedback © 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

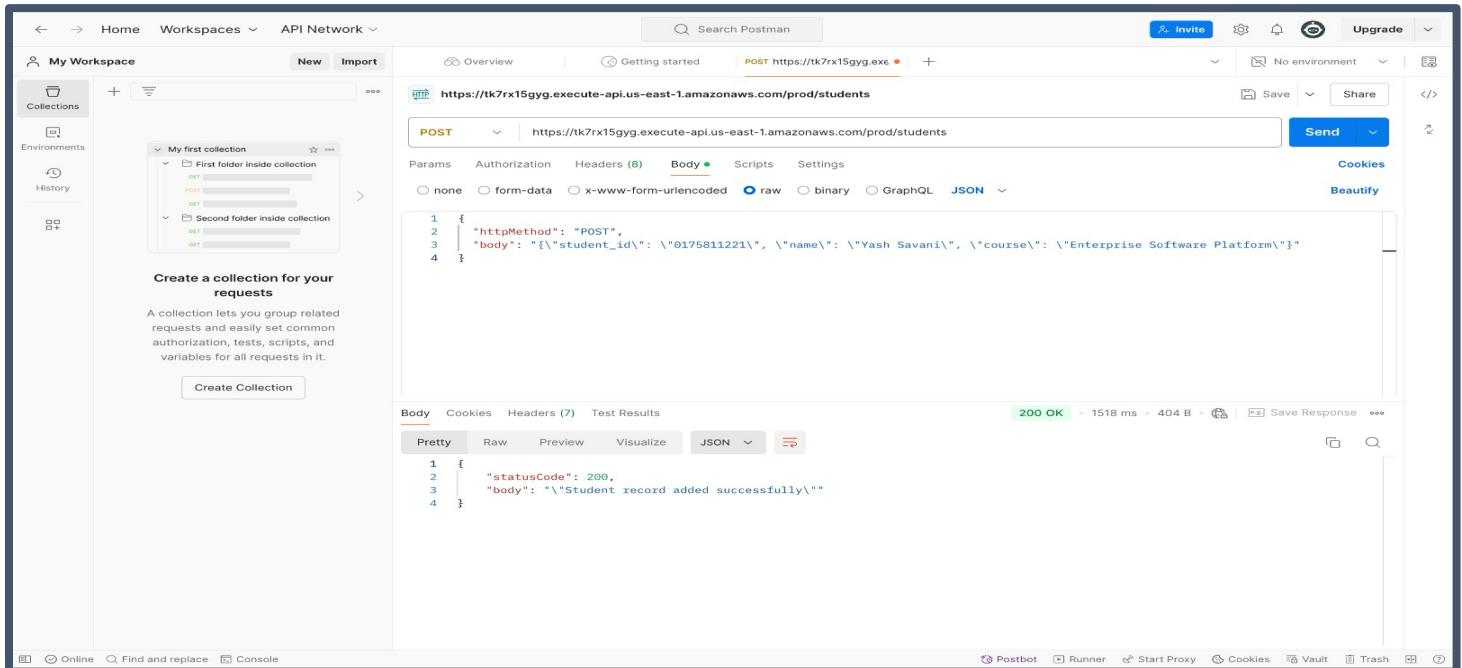
The screenshot shows the AWS API Gateway console with the URL <https://us-east-1.console.aws.amazon.com/apigateway/main/apis/tk7rx15gyg/resources?api=tk7rx15gyg&region=us-east-1>. A green banner at the top indicates "Successfully created method 'PUT' in 'students'. Redeploy your API for the update to take effect." The left sidebar shows the API: StudentAPI with Resources selected. The main panel displays the /students resource with a PUT method selected. The ARN is listed as arn:aws:execute-api:us-east-1:588738571984:tk7rx15gyg/\*PUT/students. The Method request settings show Authorization set to NONE and Request validator set to False. The Integration request settings show API key required set to False and SDK operation name set to None. The Integration response settings show Method response set to Test. The Deploy API button is visible in the top right corner.

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:



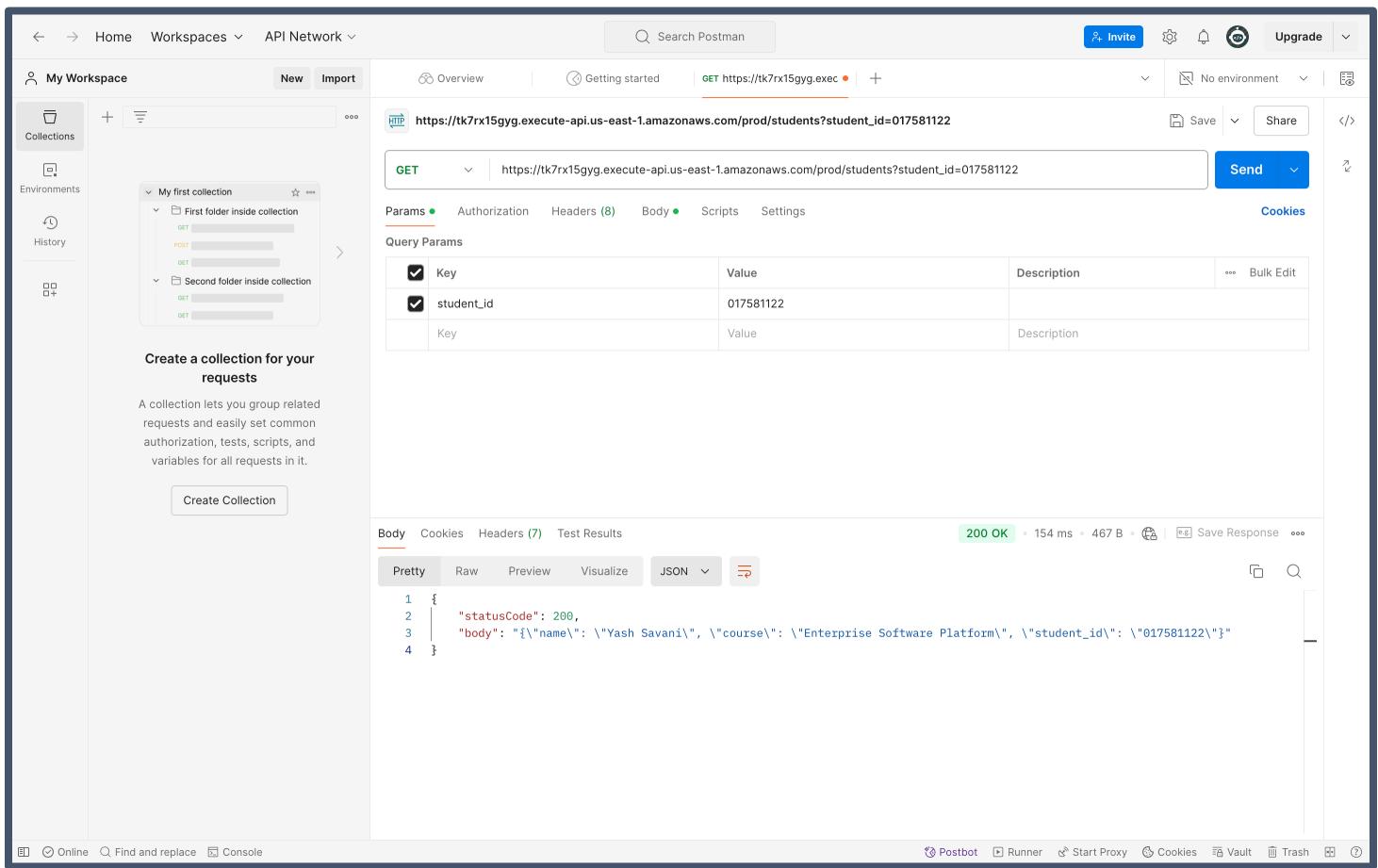
The screenshot shows the Postman interface with a POST request to `https://tk7rx15ggy.execute-api.us-east-1.amazonaws.com/prod/students`. The request body is set to raw JSON:

```
1 {  
2   "httpMethod": "POST",  
3   "body": "{\"student_id\": \"0175811221\", \"name\": \"Yash Savani\", \"course\": \"Enterprise Software Platform\"}"  
4 }
```

The response status is 200 OK with a response body of:

```
1 {  
2   "statusCode": 200,  
3   "body": "\"Student record added successfully\""  
4 }
```

## GET Method:



The screenshot shows the Postman interface with a GET request to `https://tk7rx15ggy.execute-api.us-east-1.amazonaws.com/prod/students?student_id=017581122`. The request includes a query parameter `student_id` with value `017581122`.

The response status is 200 OK with a response body of:

```
1 {  
2   "statusCode": 200,  
3   "body": "{\"name\": \"Yash Savani\", \"course\": \"Enterprise Software Platform\", \"student_id\": \"017581122\"}"  
4 }
```

## PUT Method:

The screenshot shows the Postman application interface. In the top navigation bar, 'PUT https://tk7rx15ggy.execute-api.us-east-1.amazonaws.com/prod/students?student\_id=017581122' is selected. The 'Body' tab is active, displaying the following JSON payload:

```
1 {
2   "httpMethod": "PUT",
3   "queryStringParameters": {
4     "student_id": "017581122"
5   },
6   "body": "{\"name\": \"Smit Savani\", \"course\": \"Enterprise Software Platform\"}"
7 }
```

Below the body, the response status is 200 OK with a response time of 1535 ms and a response size of 406 B. The response body is:

```
1 {
2   "statusCode": 200,
3   "body": "\"Student record updated successfully\""
4 }
```

## DELETE Method:

The screenshot shows the Postman application interface. In the top navigation bar, 'DEL https://tk7rx15ggy.execute-api.us-east-1.amazonaws.com/prod/students?student\_id=017581122' is selected. The 'Body' tab is active, displaying the following JSON payload:

```
1 {
2   "httpMethod": "DELETE",
3   "queryStringParameters": {
4     "student_id": "017581122"
5   }
6 }
```

Below the body, the response status is 200 OK with a response time of 822 ms and a response size of 406 B. The response body is:

```
1 {
2   "statusCode": 200,
3   "body": "\"Student record deleted successfully\""
4 }
```

## **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.