Development of a Serverless Web Application with AWS

Academic Report

Prepared by:

Kenza Elhamchi
Zakaria Benlamkadam

Supervised by:

Hayat Routaib

December 2024

National School of Applied Sciences Al-Hoceima

Contents

1 Introduction

1.1 Project Context and Objective

Modern businesses aim to develop web solutions that are cost-effective, high-performing, and scalable. Serverless architectures offer an attractive alternative to traditional systems, where the cloud provider manages infrastructure, scaling, and resource allocation automatically. This eliminates the need for server management and reduces infrastructure costs, allowing businesses to focus on development rather than operations.

The objective of this project is to build a web application for recording and retrieving student data using AWS serverless technologies. The solution is designed to be cost-efficient, scalable, and easy to maintain.

1.2 Importance of Serverless Architectures

Serverless architecture simplifies development by removing the complexity of infrastructure management. Key benefits include:

- **Elimination of Server Management:** No need for physical or virtual server management; infrastructure is handled by the cloud provider.
- **Pay-as-you-go Billing:** Costs are based on actual resource usage, reducing unnecessary expenses.
- **Automatic Scalability:** Resources automatically scale based on workload demands.
- **Rapid Development:** Services like AWS Lambda reduce development time with simplified configurations.

These advantages make serverless a perfect fit for applications requiring flexibility and cost-effectiveness.

1.3 Overview of Tools Used

The project uses the following AWS services:

- **Amazon S3:** Hosts the frontend (static files like HTML, CSS, and JavaScript).
- **Amazon CloudFront:** Distributes the static content hosted on S3 globally, reducing latency by caching the content at edge locations closer to users, ensuring faster load times and improved performance.
- **AWS Lambda:** Executes backend code in response to API requests without managing servers.
- **Amazon API Gateway:** Manages HTTP requests to Lambda functions.
- **Amazon DynamoDB:** A NoSQL database for storing student data.

By integrating these tools, we developed a serverless web application that is scalable, cost-efficient, and easy to maintain.

2 Project Description

2.1 Project Objective

This project demonstrates how serverless architectures can address the scalability, cost, and management challenges in modern web applications. The main focus is on simplifying the management of student data while minimizing infrastructure costs. By using a serverless setup, this project eliminates backend infrastructure concerns and ensures seamless user experiences even during traffic surges.

2.1.1 Problems Addressed:

- **Server Management Complexity:** Traditional systems require constant monitoring and management of infrastructure, which is resolved with serverless services.
- **Scaling Challenges:** Traditional systems may not scale efficiently during traffic spikes. The serverless architecture automatically adapts to demand.
- **Data Management Efficiency:** Serverless solutions, like DynamoDB, provide low-latency access to data and scale automatically.

2.2 Use Cases

The application includes the following core features:

- **Storing Student Data:** Users submit student details (e.g., name, class), which are processed and saved in DynamoDB through Lambda functions.
- **Retrieving Student Data:** Users can search for stored student information by providing an identifier or search parameter.
- **Optional Data Modification:** Features to update or delete student data may be added in future iterations.
- **Frontend Hosting:** The user interface is hosted on AWS S3 as a static website.
- **Request Management:** API Gateway handles secure requests, with optional authentication mechanisms.

2.3 System Architecture

2.3.1 Overview

Global Architecture Diagram
Figure 20 illustrates the architecture of the system, showcasing the flow of data between components:

- **Frontend:** Hosted on AWS S3 and delivered through Amazon CloudFront. CloudFront caches static content (HTML, CSS, JavaScript, etc.) at edge locations to ensure faster load times and improved performance globally.
- **API Gateway:** Manages incoming HTTP requests.

- **Lambda:** Handles business logic and interacts with DynamoDB.
- **DynamoDB:** Stores and retrieves data.

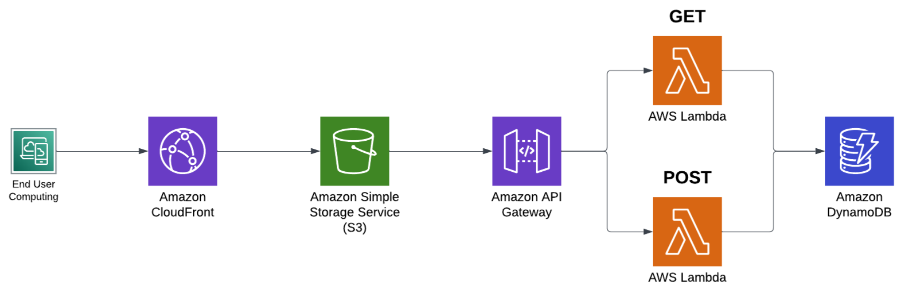


Figure 1: Global Architecture Diagram

Explanation of the Main Components:
The serverless architecture leverages AWS services to minimize infrastructure management and ensure scalability:

- **Frontend:** Hosted as static files in an S3 bucket and delivered through Amazon CloudFront.
- **Backend:** The backend is serverless, with requests routed via API Gateway to Lambda functions, which then interact with DynamoDB.
- **Database:** DynamoDB stores application data efficiently, ensuring fast reads and writes.

2.3.2 Role of Each Component

S3 and CloudFront: Hosting the Application

- S3 stores the frontend (HTML, CSS, JavaScript).
- CloudFront is used to distribute the static content globally, caching it at edge locations to reduce latency and improve load times for end users.
- CloudFront serves the static files from S3 to the browser, ensuring high availability and minimal latency.

Benefits:

- Scales automatically and charges only for storage and data transfer. CloudFront further optimizes the delivery of static content by reducing latency and speeding up access for global users.

API Gateway: Managing HTTP Requests

- Acts as the entry point for HTTP requests from the frontend.
- Routes requests to Lambda functions based on the HTTP method (GET, POST, PUT, DELETE).

Benefits:

- Secure communication, request validation, and monitoring features are available.

Lambda: Executing Backend Code

- Lambda functions execute the application logic, such as validating inputs and interacting with DynamoDB.
- They are triggered by requests via API Gateway.

Benefits:

- Pay-per-use, automatic scaling, and high availability.

DynamoDB: Storing Data

- A NoSQL database for storing student records.

Features:

- High throughput, low latency, and automatic scaling for large datasets. Security is ensured with encryption at rest.

3 Technical Implementation

3.1 1. Frontend Development

The frontend of the application is developed using HTML, CSS, and JavaScript files. The goal is to create a simple and interactive user interface that allows interaction with the backend by submitting and retrieving data.

3.1.1 1.1 Website Structure

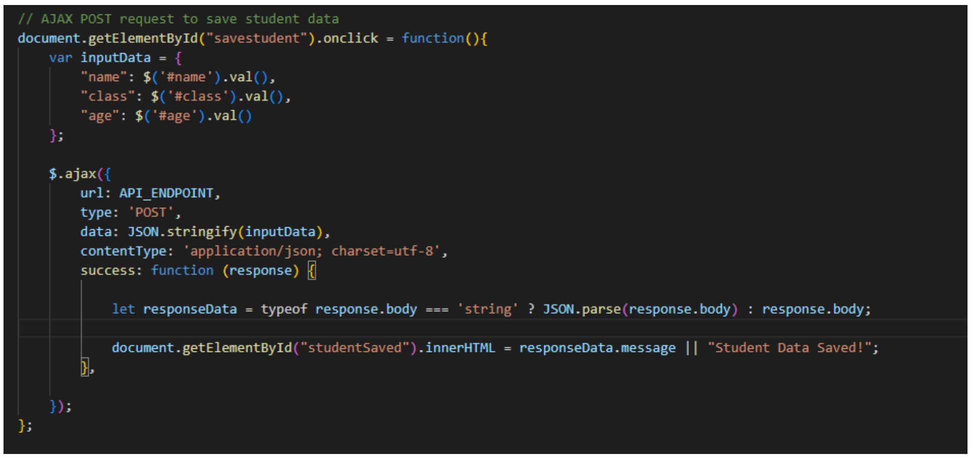
The user interface is organized to be clear and user-friendly. Here is the website structure:

- `index.html`: This is the main file that contains the structure of the homepage, including input fields for student information and buttons to interact with the API.
- `scripts.js`: This JavaScript file is essential for managing interactions between the frontend and backend. It allows sending HTTP requests to APIs and receiving responses.

3.1.2 1.2 Description of HTML, CSS, and JavaScript Files Used

- **HTML (index.html)**: The HTML file contains forms for submitting student information and a button to fetch data. It is designed to be responsive and suitable for different screen sizes. the css included inside the index.html
- **JavaScript (scripts.js)**: This file manages the logic for API calls. For example, it uses `fetch()` to send POST and GET requests to the API Gateway.

Example Code for API Call in scripts.js:



```javascript
// AJAX POST request to save student data
document.getElementById("savestudent").onclick = function(){
    var inputData = {
        "name": $('#name').val(),
        "class": $('#class').val(),
        "age": $('#age').val()
    };

    $.ajax({
        url: API_ENDPOINT,
        type: 'POST',
        data: JSON.stringify(inputData),
        contentType: 'application/json; charset=utf-8',
        success: function (response) {

            let responseData = typeof response.body === 'string' ? JSON.parse(response.body) : response.body;

            document.getElementById("studentSaved").innerHTML = responseData.message || "Student Data Saved!";
        },

    });
};
```

Figure 2: Example Code for API Call in scripts.js

In this code, the `addStudent()` function sends a POST request to the API Gateway to save a student's data in DynamoDB.

Image to Insert: Screenshot of the JavaScript code (scripts.js).

3.2 2. Backend Development

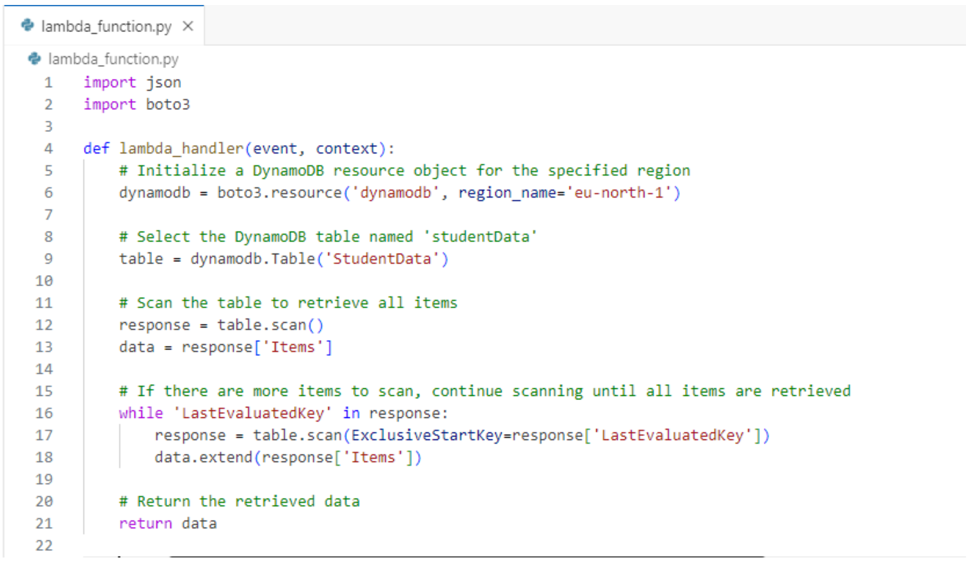
The backend of the application is fully managed by AWS Lambda functions, enabling automatic resource management without server infrastructure concerns. Each Lambda function is triggered by HTTP calls through the API Gateway.

3.2.1 2.1 Lambda Function Definitions

The two main Lambda functions used in this project are:

- **GET Function**: This function retrieves a student's information based on their ID.

Example Code for GET Function:

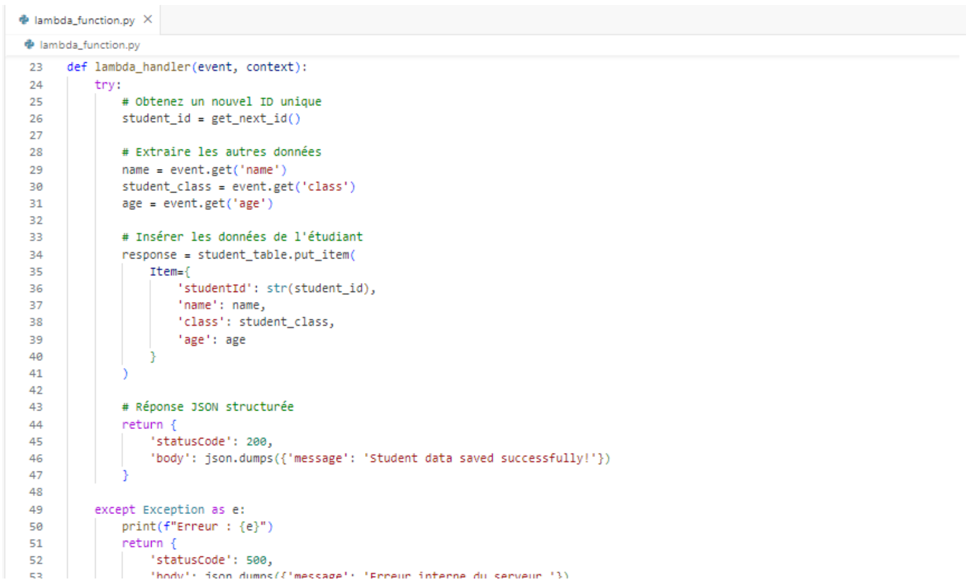


```python
import json
import boto3

def lambda_handler(event, context):
    # Initialize a DynamoDB resource object for the specified region
    dynamodb = boto3.resource('dynamodb', region_name='eu-north-1')

    # Select the DynamoDB table named 'studentData'
    table = dynamodb.Table('StudentData')

    # Scan the table to retrieve all items
    response = table.scan()
    data = response['Items']

    # If there are more items to scan, continue scanning until all items are retrieved
    while 'LastEvaluatedKey' in response:
        response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
        data.extend(response['Items'])

    # Return the retrieved data
    return data

```

Figure 3: Example Code for GET Function

- **POST Function**: This function saves new data in DynamoDB.

Example Code for POST Function:



lambda_function.py

```python
def lambda_handler(event, context):
    try:
        # Obtenez un nouvel ID unique
        student_id = get_next_id()

        # Extraire les autres données
        name = event.get('name')
        student_class = event.get('class')
        age = event.get('age')

        # Insérer les données de l'étudiant
        response = student_table.put_item(
            Item={
                'studentId': str(student_id),
                'name': name,
                'class': student_class,
                'age': age
            }
        )

        # Réponse JSON structurée
        return {
            'statusCode': 200,
            'body': json.dumps({'message': 'Student data saved successfully!'})
        }

    except Exception as e:
        print(f"Erreur : {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({'message': 'Erreur interne du serveur.'})
```

Figure 4: Example Code for POST Function

3.2.2 2.2 Description of Interactions with DynamoDB

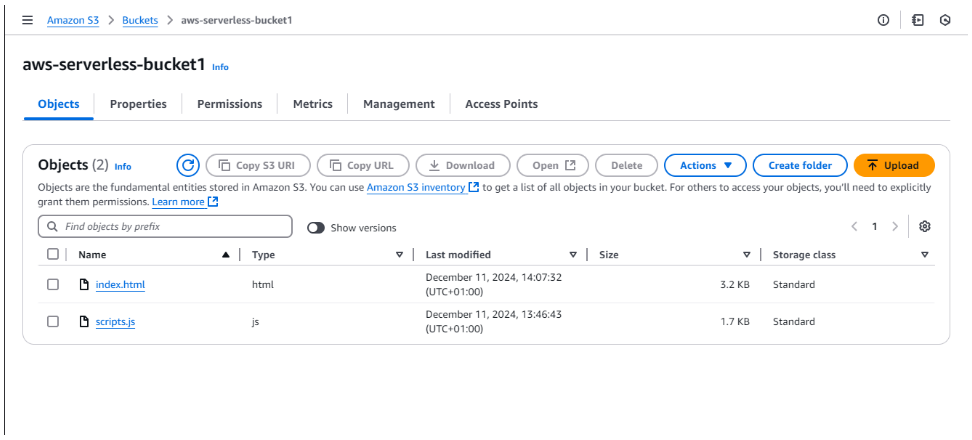
DynamoDB is used to store student information. Each student is identified by a unique `student_id`. When the POST function is called, the data is sent to DynamoDB where a new item is added. When the GET function is called, DynamoDB searches for the student corresponding to the provided ID and returns the relevant information.

3.3 3. AWS Configuration

This section details the steps to configure the necessary AWS services for this project: S3, API Gateway, Lambda, and DynamoDB.

3.3.1 3.1 Steps to Configure S3 (Static Hosting)

1. Log in to the AWS Management Console.
2. Navigate to S3 and create a new bucket.
3. Upload the HTML, and JavaScript files to the bucket.
4. Enable the static hosting option for the bucket.
5. Configure the bucket to be publicly accessible so that the application can be accessed via a URL.


Amazon S3 > Buckets > aws-serverless-bucket1

aws-serverless-bucket1 Info

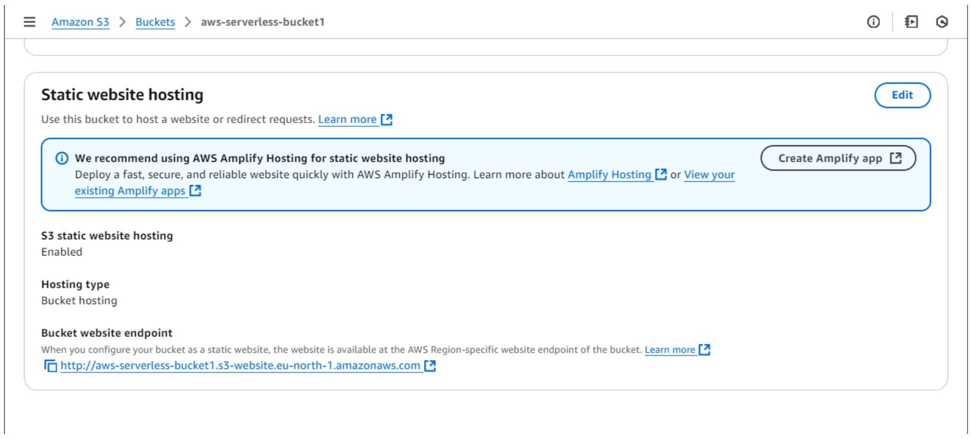
Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (2) Info

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
index.html	html	December 11, 2024, 14:07:32 (UTC+01:00)	3.2 KB	Standard
scripts.js	js	December 11, 2024, 13:46:43 (UTC+01:00)	1.7 KB	Standard

Figure 5: Bucket S3


Amazon S3 > Buckets > aws-serverless-bucket1

Static website hosting

Use this bucket to host a website or redirect requests. Learn more

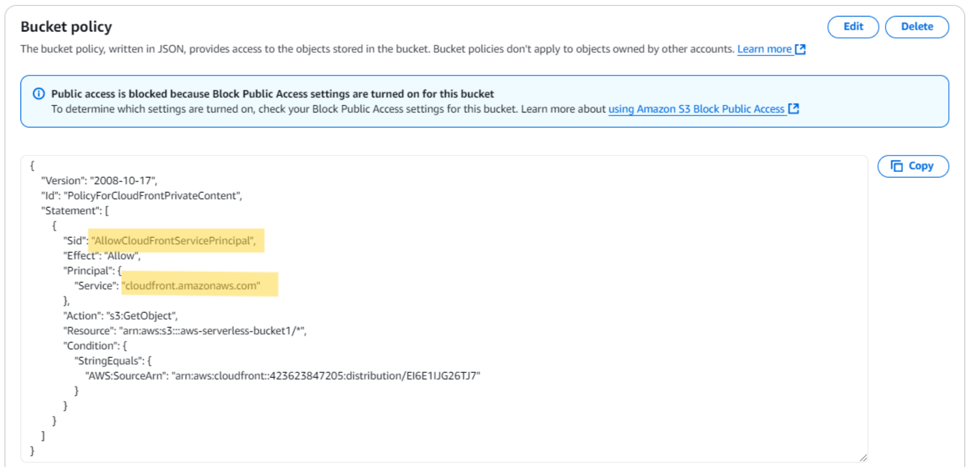
We recommend using AWS Amplify Hosting for static website hosting
Deploy a fast, secure, and reliable website quickly with AWS Amplify Hosting. Learn more about Amplify Hosting or View your existing Amplify apps

S3 static website hosting
Enabled

Hosting type
Bucket hosting

Bucket website endpoint
When you configure your bucket as a static website, the website is available at the AWS Region-specific website endpoint of the bucket. Learn more
http://aws-serverless-bucket1.s3-website.eu-north-1.amazonaws.com

Figure 6: Enable Web Hosting in Bucket S3


Bucket policy

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owned by other accounts. Learn more

Public access is blocked because Block Public Access settings are turned on for this bucket
To determine which settings are turned on, check your Block Public Access settings for this bucket. Learn more about using Amazon S3 Block Public Access

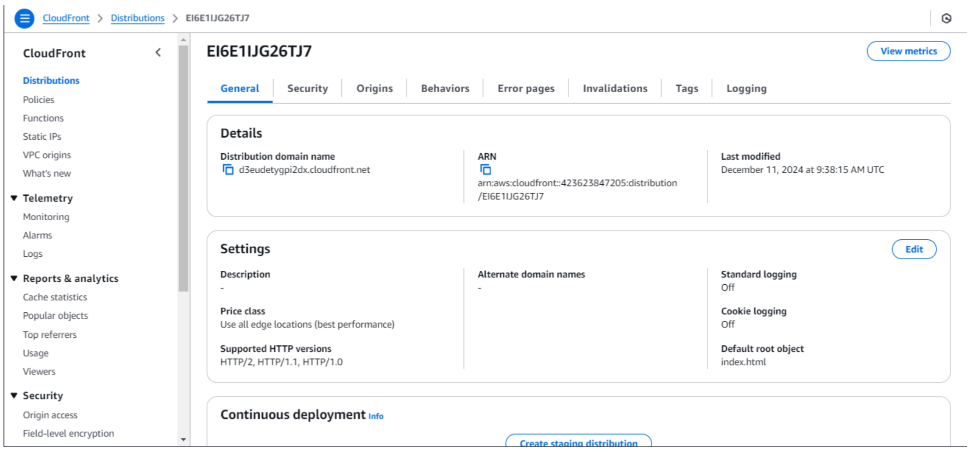
```
{
  "Version": "2008-10-17",
  "Id": "PolicyForCloudFrontPrivateContent",
  "Statement": [
    {
      "Sid": "AllowCloudFrontServicePrincipal",
      "Effect": "Allow",
      "Principal": {
        "Service": "cloudfront.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::aws-serverless-bucket1/*",
      "Condition": {
        "StringEquals": {
          "AWS:SourceArn": "arn:aws:cloudfront::423623847205:distribution/EI6E1IJG26TJ7"
        }
      }
    }
  ]
}
```

Figure 7: Bucket Policy

3.3.2 3.2 Steps to Configure CloudFront

1. Log in to the AWS Management Console and navigate to CloudFront.

2. Click on **Create Distribution**.
3. Under the **Origin Settings**, select the S3 bucket created earlier as the origin.
4. Enable **Origin Access Control (OAC)** to secure the connection between CloudFront and the S3 bucket.
5. Specify the default root object, such as `index.html`, so CloudFront knows which file to serve.
6. Modify the S3 bucket policy to allow CloudFront access by including the CloudFront service principal (`cloudfront.amazonaws.com`) in the policy.
7. Block all public access to the S3 bucket to prevent direct exposure.
8. Click on **Create Distribution** and wait for the distribution to deploy (this may take several minutes).
9. Once the deployment is complete, use the CloudFront-provided secure **HTTPS URL** to access your application.



EI6E1IJG26TJ7

Details

Distribution domain name
d3eudetygpi2dx.cloudfront.net

ARN
arn:aws:cloudfront::423623847205:distribution/EI6E1IJG26TJ7

Last modified
December 11, 2024 at 9:38:15 AM UTC

Settings

Description
-

Alternate domain names
-

Standard logging
Off

Price class
Use all edge locations (best performance)

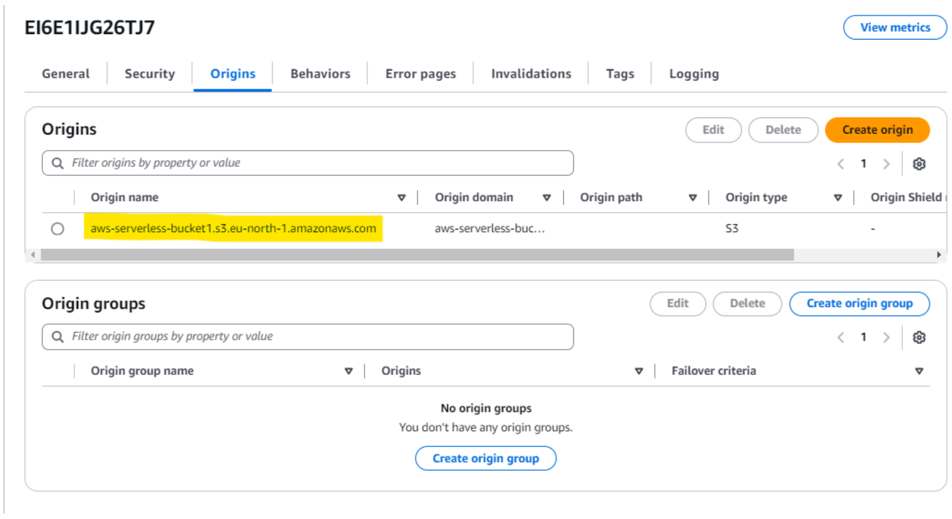
Cookie logging
Off

Supported HTTP versions
HTTP/2, HTTP/1.1, HTTP/1.0

Default root object
index.html

Continuous deployment Info

Figure 8: cloudfront distribution



EI6E1IJG26TJ7

Origins

Origin name	Origin domain	Origin path	Origin type	Origin Shield
aws-serverless-bucket1.s3.eu-north-1.amazonaws.com	aws-serverless-buc...		S3	-

Origin groups

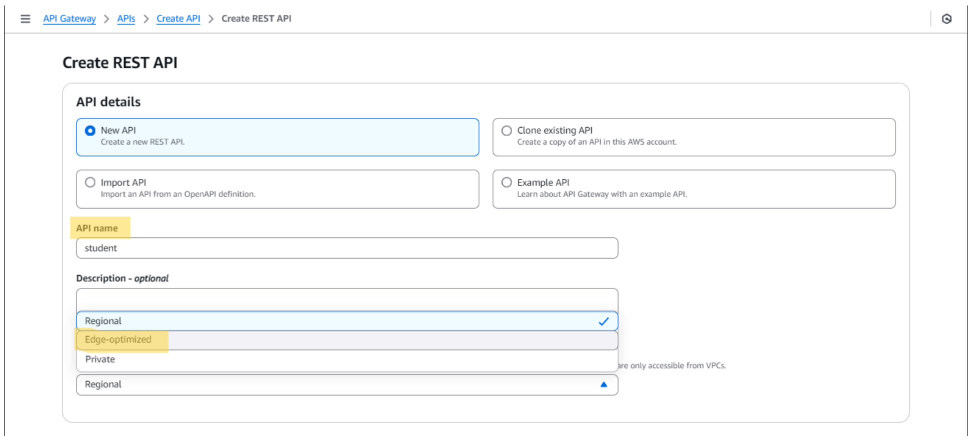
Origin group name	Origins	Failover criteria

No origin groups
You don't have any origin groups.

Figure 9: resource origins

3.3.3 3.3 Creating APIs with API Gateway

1. Navigate to API Gateway and create a new REST API.
2. Define resources and methods (GET, POST) to interact with the Lambda functions.
3. Link each API method to its corresponding Lambda function.
4. Configure permissions so that only authorized requests can access the API.


API Gateway > APIs > Create API > Create REST API

Create REST API

API details

- New API — Create a new REST API.
- Clone existing API — Create a copy of an API in this AWS account.
- Import API — Import an API from an OpenAPI definition.
- Example API — Learn about API Gateway with an example API.

API name: student

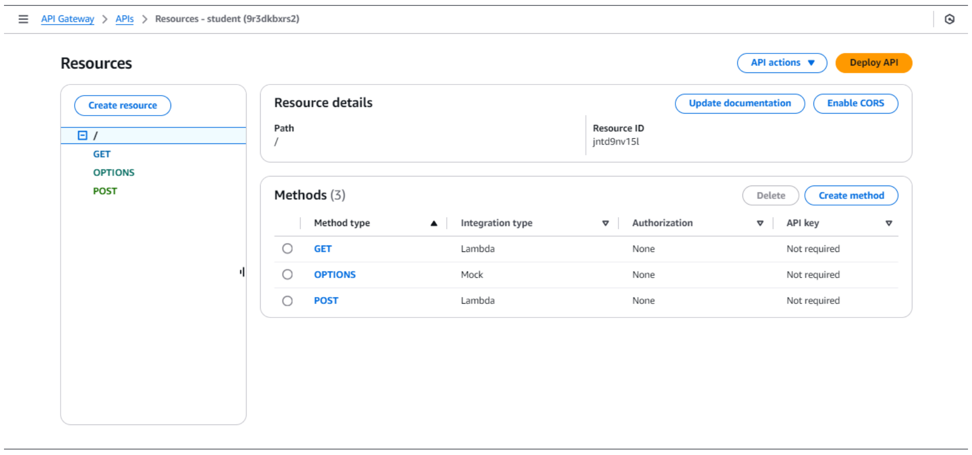
Description - *optional*

Regional ✓
Edge-optimized
Private

…are only accessible from VPCs.

Regional

Figure 10: Create Rest API


API Gateway > APIs > Resources - student (9r3dkbxrs2)

Resources

/
GET
OPTIONS
POST

Resource details

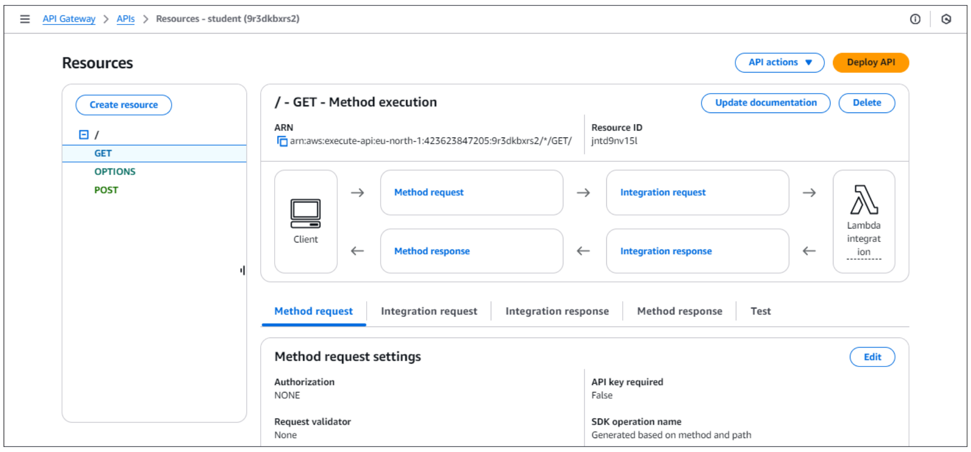
Path: /

Resource ID: jntd9nv15l

Methods (3)

Method type	Integration type	Authorization	API key
GET	Lambda	None	Not required
OPTIONS	Mock	None	Not required
POST	Lambda	None	Not required

Figure 11: API Resources


API Gateway > APIs > Resources - student (9r3dkbxrs2)

Resources

API actions ▼ | Deploy API

Create resource

/
GET
OPTIONS
POST

/ - GET - Method execution

Update documentation | Delete

ARN
arn:aws:execute-api:eu-north-1:423623847205:9r3dkbxrs2/*/GET/

Resource ID
jntd9nv15l

Method request | Integration request | Integration response | Method response | Test

Method request settings

Edit

Authorization
NONE

API key required
False

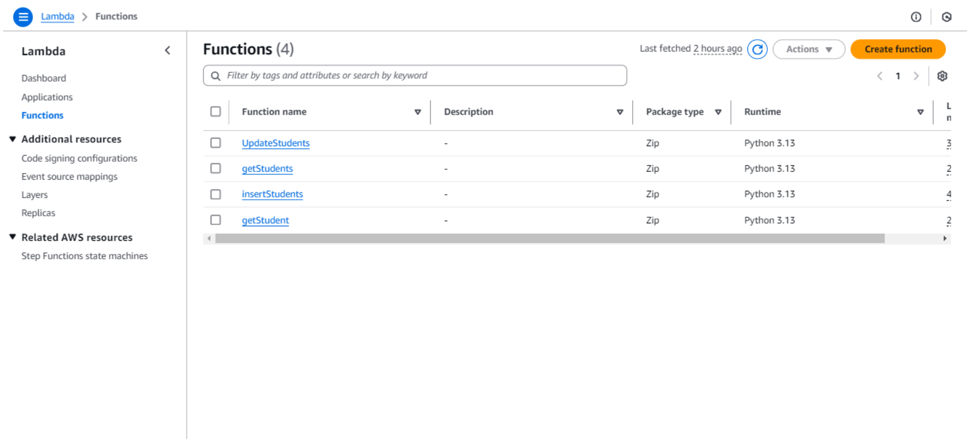
Request validator
None

SDK operation name
Generated based on method and path

Figure 12: Example Get Resource

3.3.4 3.4 Deploying Lambda Functions

1. Go to AWS Lambda and create two functions: one for GET and one for POST.
2. Add the necessary permissions for Lambda functions to access DynamoDB.
3. Deploy the Lambda functions to make them accessible via API Gateway.


Lambda > Functions

Lambda

Dashboard
Applications
Functions

▼ Additional resources
Code signing configurations
Event source mappings
Layers
Replicas

▼ Related AWS resources
Step Functions state machines

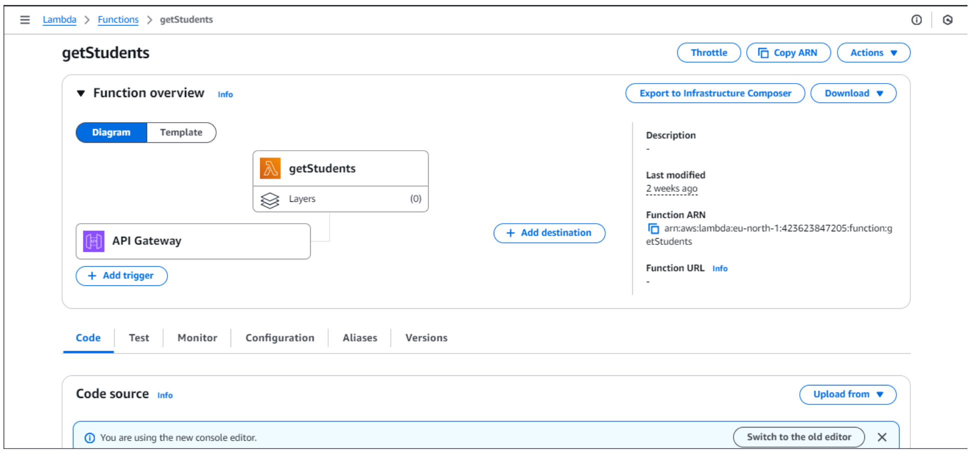
Functions (4)

Last fetched 2 hours ago | Actions ▼ | Create function

Filter by tags and attributes or search by keyword

Function name	Description	Package type	Runtime
UpdateStudents	-	Zip	Python 3.13
getStudents	-	Zip	Python 3.13
insertStudents	-	Zip	Python 3.13
getStudent	-	Zip	Python 3.13

Figure 13: lambda funtions



Description
-

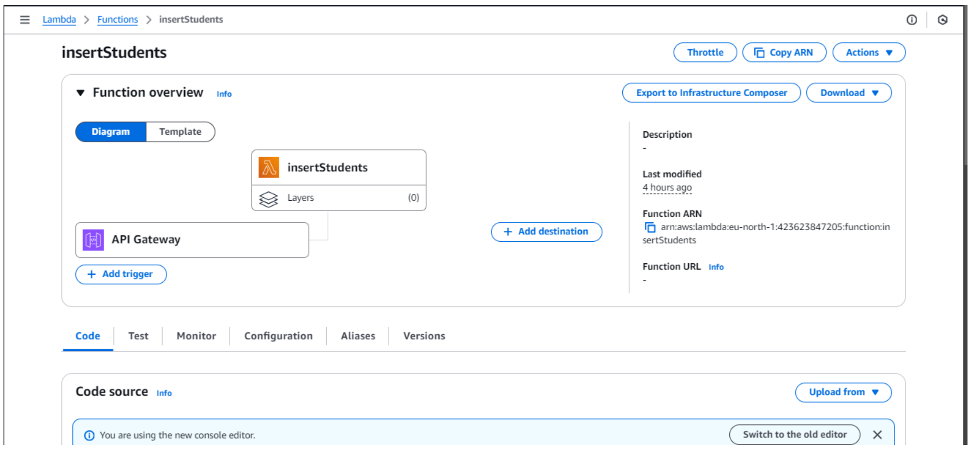
Last modified
2 weeks ago

Function ARN
arn:aws:lambda:eu-north-1:423623847205:function:getStudents

Function URL Info
-

You are using the new console editor.

Figure 14: GET FUCNTION



Description
-

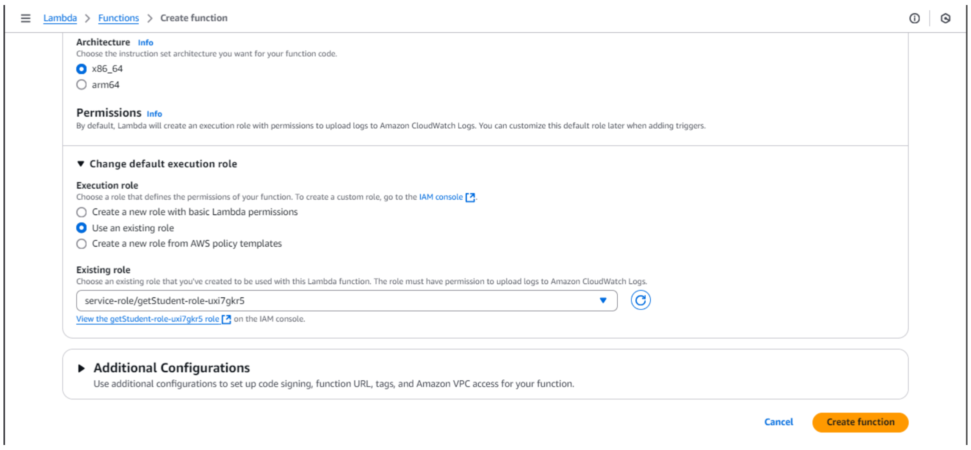
Last modified
4 hours ago

Function ARN
arn:aws:lambda:eu-north-1:423623847205:function:insertStudents

Function URL Info
-

You are using the new console editor.

Figure 15: POST FUNCTION



Architecture Info
Choose the instruction set architecture you want for your function code.
[✓ x86_64]
arm64

Permissions Info
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.

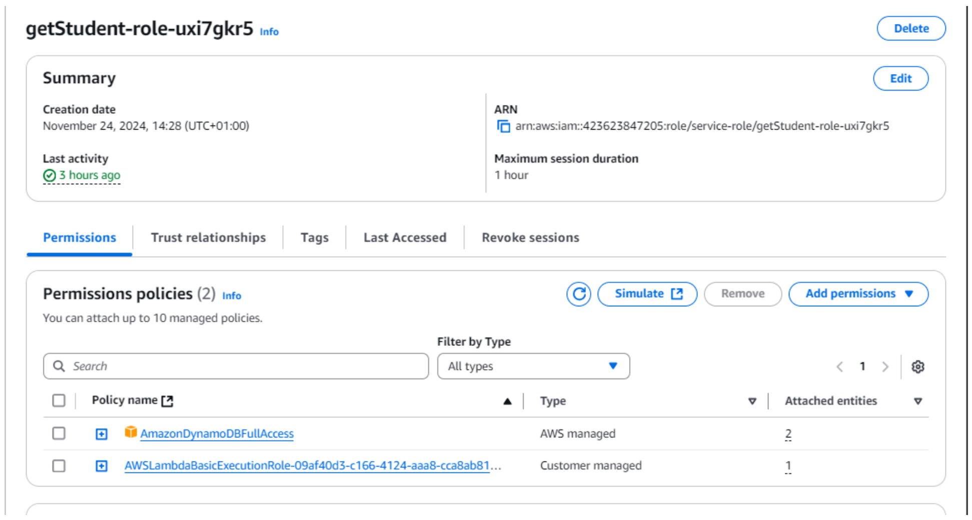
▼ Change default execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.
Create a new role with basic Lambda permissions
[✓ Use an existing role]
Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.
service-role/getStudent-role-uxi7gkr5
View the getStudent-role-uxi7gkr5 role on the IAM console.

▶ Additional Configurations
Use additional configurations to set up code signing, function URL, tags, and Amazon VPC access for your function.

Figure 16: Permissions Lambda



getStudent-role-uxi7gkr5 Info

Summary

Creation date
November 24, 2024, 14:28 (UTC+01:00)

ARN
arn:aws:iam::423623847205:role/service-role/getStudent-role-uxi7gkr5

Last activity
3 hours ago

Maximum session duration
1 hour

Permissions | Trust relationships | Tags | Last Accessed | Revoke sessions

Permissions policies (2) Info
You can attach up to 10 managed policies.

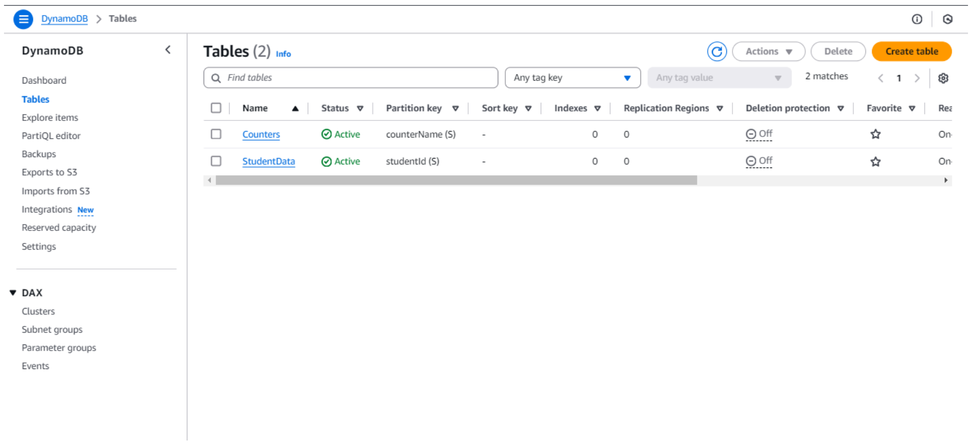
Filter by Type: All types

Policy name	Type	Attached entities
AmazonDynamoDBFullAccess	AWS managed	2
AWSLambdaBasicExecutionRole-09af40d3-c166-4124-aaa8-cca8ab81…	Customer managed	1

Figure 17: IAM ROLE

3.3.5 3.5 Configuring DynamoDB (Table and Data Schema)

1. Create a table in DynamoDB with `student_id` as the primary key.
2. Define a flexible schema to store student information (age , class).



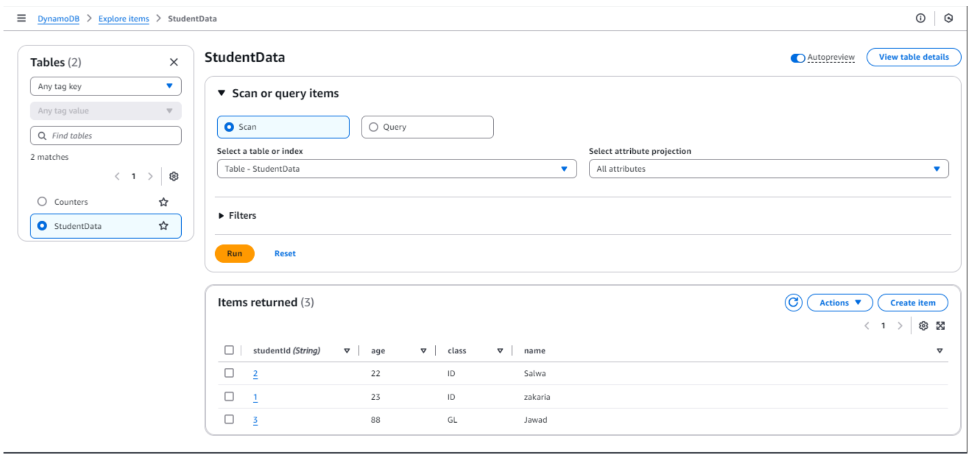
DynamoDB > Tables

Tables (2) Info

2 matches

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Rea
Counters	Active	counterName (S)	-	0	0	Off	☆	On
StudentData	Active	studentId (S)	-	0	0	Off	☆	On

Figure 18: Dynamo DB tables


DynamoDB > Explore items > StudentData

Tables (2)

Any tag key

Any tag value

Find tables

2 matches

Counters

[✓ StudentData]

StudentData

Scan or query items

[✓ Scan] Query

Select a table or index: Table - StudentData

Select attribute projection: All attributes

Filters

Items returned (3)

studentId *(String)*	age	class	name
2	22	ID	Salwa
1	23	ID	zakaria
3	88	GL	Jawad

Figure 19: Explore Items

4 Data Flow

This section describes the data flow from the user submitting a request to storing it in DynamoDB.

4.0.1 4.1 Step-by-Step Explanation of a User Request

When a user submits a form to add a student:

1. **Request Submission:** The user submits a form on the frontend (delivered via CloudFront, which serves the static website from the S3 bucket).
2. **API Request Sent:** The frontend sends an HTTP POST request to the API Gateway with the student's data.
3. **Processing by Lambda:** API Gateway invokes the POST Lambda function.
4. **Storage in DynamoDB:** The Lambda function saves the data in DynamoDB.
5. **Response to User:** Lambda sends a response to the API Gateway, which relays it to the frontend, confirming successful data storage. The response is displayed securely to the user via CloudFront.

5 Project Benefits and Limitations

5.1 4.1 Benefits

The project offers several notable advantages that contribute to its efficiency, cost-effectiveness, and ease of use.

- **Low Cost with Serverless Architecture:** Using a serverless architecture with AWS Lambda, DynamoDB, and API Gateway significantly reduces infrastructure costs. There is no need to manage servers or purchase expensive hardware resources. The pay-as-you-go model ensures we only pay for the resources actually used, making the project more affordable.

- **Automatic Scalability:** The serverless architecture enables automatic scalability. As the number of users grows, AWS automatically adjusts the capacity of services like Lambda and DynamoDB to handle the load without requiring manual intervention. This scalability ensures optimal performance even during sudden traffic spikes.
- **Ease of Deployment:** Tools like AWS Lambda and API Gateway enable quick and easy deployment. Code is deployed directly to Lambda, and APIs are configured and managed via API Gateway. This approach simplifies the production process and reduces deployment-related errors.

5.2 4.2 Limitations

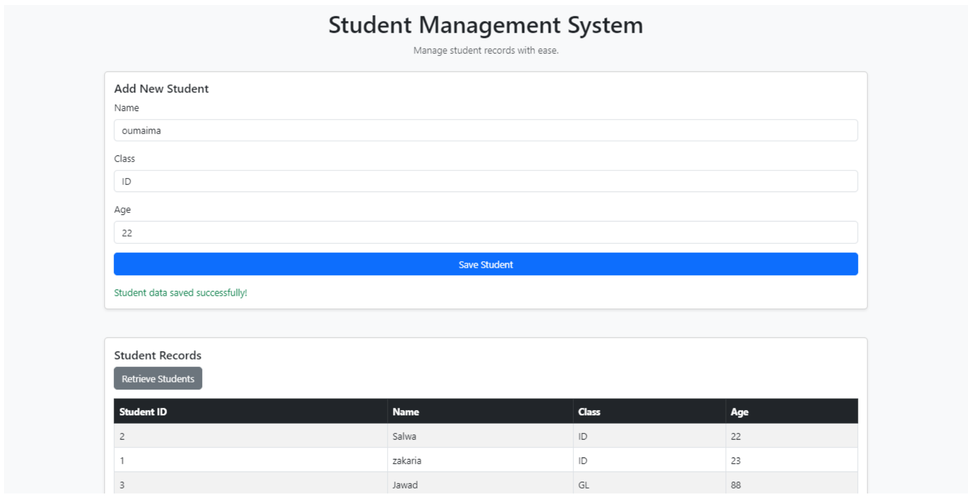
Despite its many benefits, the project has certain limitations that need consideration.

- **Dependency on AWS:** The exclusive use of AWS services like Lambda and DynamoDB creates a strong dependency on Amazon's platform. Technical issues or cost increases from AWS could directly impact the project's availability and profitability. Additionally, this dependency limits flexibility in choosing tools and providers.
- **Possible API Call Latency:** Although AWS Lambda and API Gateway offer high performance, latency can sometimes occur during API calls, especially if the Lambda service hasn't been invoked recently (cold start). This latency may affect user experience, particularly in applications requiring real-time responses.

6 Results

The project outcomes include the setup of a functional and efficient application to manage information using a serverless architecture.

- **Screenshots of the Website:** Below are screenshots of the functional website, showcasing the user interface and application usability.


Student Management System

Manage student records with ease.

Add New Student

Name: oumaima

Class: ID

Age: 22

Student data saved successfully!

Student Records

Student ID	Name	Class	Age
2	Salwa	ID	22
1	zakaria	ID	23
3	Jawad	GL	88

Figure 20: Screenshot of the app

- **Example of Data Stored in DynamoDB:** Here is an example of the data stored in the DynamoDB table for a student.

```
{
    "student_id": "12345",
    "name": "John Doe",
    "class":"Math",
    "Age":"23"
}
```

- **Tests Conducted to Verify Functionality:** Tests were conducted to validate the functionality of the website and APIs. These tests included data submissions via the form and verification of data storage in DynamoDB. All tests showed that data was correctly recorded and retrieved.

7 Conclusion and Future Work

This project demonstrated the effectiveness of serverless architecture for managing student information. By leveraging AWS services such as Lambda, DynamoDB, and API Gateway, the application proved to be performant, scalable, and cost-efficient.

- **Project Summary and Results:** The project successfully implemented a system that enables submitting and retrieving information via a static frontend website hosted on S3 and backend AWS Lambda functions. Data was correctly processed and stored in DynamoDB, and the application showcased the advantages of a serverless environment.
- **Suggestions for Improving the Project:**
  - Add Authentication: Integrate AWS Cognito to ensure only authorized users can submit or retrieve information, improving security.
  - Enhance Logging and Monitoring: Use AWS CloudWatch to monitor application performance and troubleshoot issues efficiently.
  - Implement Version Control: Use AWS CodePipeline to automate deployment and versioning of the application for continuous delivery.
  - **Data Backup and Recovery:** Use AWS Backup to automate periodic backups of DynamoDB and ensure data durability.
  - **Integration with AI Services:** Incorporate AWS SageMaker for predictive analytics or personalization features, such as suggesting courses based on student data.
  - **Event-Driven Architecture:** Use AWS EventBridge to trigger actions based on specific events, such as sending notifications for new data entries.
  - **Compliance and Security Enhancements:** Implement AWS Identity and Access Management (IAM) policies and AWS Shield to secure the application against DDoS attacks.
  - **Cost Optimization:** Use AWS Cost Explorer and Trusted Advisor to identify areas to further optimize resource usage and minimize costs.

- **Serverless CI/CD Pipeline:** Implement a fully serverless CI/CD pipeline using AWS CodeBuild and AWS CodeDeploy.