Introduction

This project showcases a fully serverless backend API built using Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. Combined with Amazon S3 and CloudFront for frontend hosting, the architecture delivers a scalable, secure, and cost-efficient solution for handling CRUD operations without managing servers. The solution offers a robust backend capable of processing user requests, executing business logic, and storing data with high scalability and minimal infrastructure administration.

Tech Stack

The serverless solution uses the following AWS services and technologies:

- **AWS Lambda:** Executes backend business logic for handling API requests with automatic scaling.
- **Amazon API Gateway:** Serves as the RESTful API layer, routing GET and POST requests to Lambda functions.
- **Amazon DynamoDB:** Stores application data in a fast, highly scalable NoSQL database with low-latency reads and writes.
- **Amazon S3:** Hosts the static frontend files (HTML, JavaScript) and provides durable object storage.
- **Amazon CloudFront:** Delivers the web application globally with low latency and improved performance using CDN caching.
- **AWS IAM:** Manages secure access permissions between Lambda, API Gateway, and DynamoDB.
- **JavaScript (Frontend):** Handles user interactions and sends requests to the API Gateway endpoint.

Prerequisites

To follow along with this project, ensure you have:

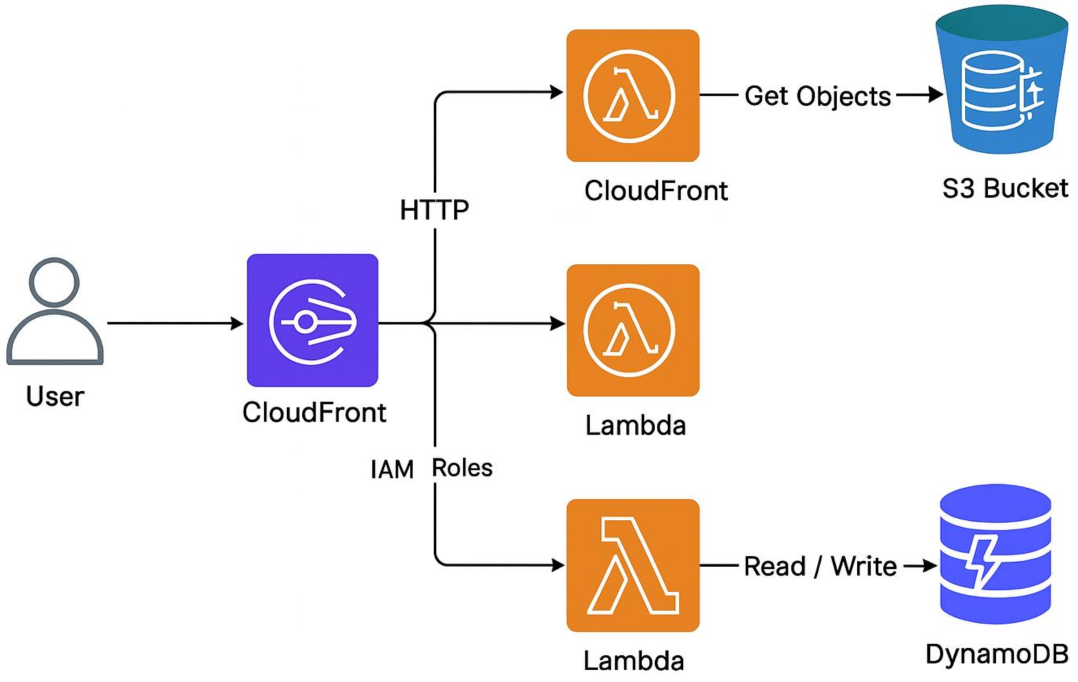
- **Basic AWS Knowledge:** Familiarity with Lambda, API Gateway, S3, DynamoDB, and IAM services.

- **AWS CLI (Optional):** Installed and configured if you want to manage or deploy resources from the command line.
- **IAM Permissions:** Proper permissions to create and manage Lambda functions, API Gateway resources, DynamoDB tables, S3 buckets, and CloudFront distributions.

Problem Statement / Use Case

A fictional organization needs a simple, scalable system to manage student records without maintaining traditional servers or databases. The goal is to build a fully serverless backend that allows users to add, retrieve, and manage student information securely and efficiently. Using AWS services such as API Gateway, AWS Lambda, DynamoDB, S3, and CloudFront, this project delivers an application that processes user requests in real-time and stores data reliably. The solution enables seamless CRUD operations while ensuring high scalability, low maintenance, and minimal operational overhead.

Architecture Diagram



Component Breakdown

Amazon API Gateway:
Acts as the entry point for all client requests, routing HTTP calls to the appropriate AWS Lambda functions. Handles throttling, request validation, and secure endpoint exposure.

AWS Lambda:
Serves as the compute layer for all backend logic. Executes CRUD operations, data validation, and business logic without requiring server management.

Amazon DynamoDB:
Provides a fully managed NoSQL database for storing user records, application data, and API responses with high performance and automatic scaling.

Amazon S3:
Stores static website content such as HTML, CSS, and JavaScript files, and also holds any application assets or uploads.

Amazon CloudFront:
Delivers the frontend application globally with low latency by caching content at edge locations, ensuring fast and secure user access.

Amazon Cognito:
Manages user authentication and authorization, enabling secure sign-up, sign-in, and token-based API access.

AWS IAM:
Defines secure permissions for Lambda functions, API Gateway, Cognito roles, and DynamoDB interactions.

AWS CloudWatch:
Monitors logs, API usage, Lambda performance, and system health, providing visibility into the entire serverless stack.

Step-by-step implementation

0) Quick assumptions & prerequisites

- You have an AWS account and AWS CLI configured (`aws configure`).
- You have basic familiarity with AWS Console.
- Region used: `us-east-1` (replace where needed).
- Node/JS or plain HTML+JS for frontend (examples use JS `fetch`).

1) Create S3 bucket and upload frontend

1. Create bucket (example CLI):

```
aws s3 mb s3://my-serverless-student-app --region us-east-1
```

2.Enable static website hosting in console or via CLI (point index document to `index.html`).

3. Upload files:

```
aws s3 sync frontend/ s3://my-serverless-student-app
```

4.Make bucket private — we'll use CloudFront to serve content; do **not** make bucket public. Configure OAC (Origin Access Control) in CloudFront later.

2) Create CloudFront distribution (pointing to S3)

- In Console → CloudFront → Create distribution:
  - Origin: S3 bucket (use OAC / origin access identity).
  - Default root object: `index.html`
  - Restrict bucket access: yes (CloudFront OAC).
- Save distribution and copy domain (e.g., `d123.cloudfront.net`) — this will be your frontend URL.

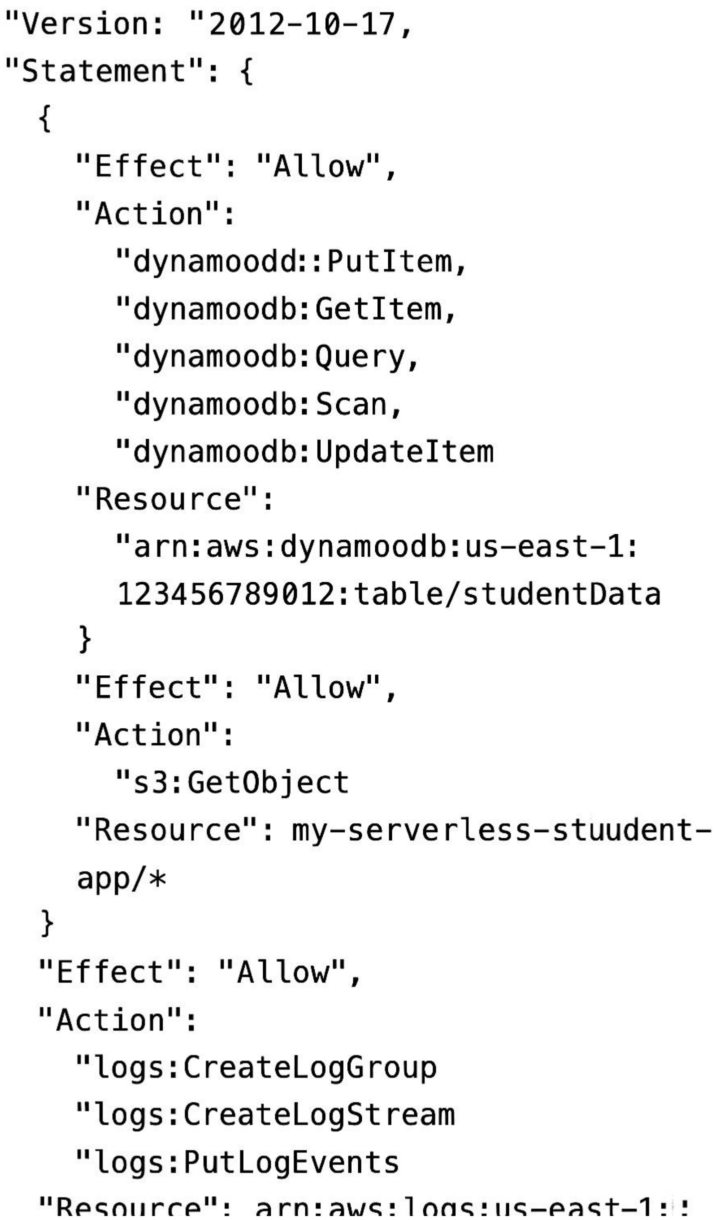
Important: Update S3 bucket policy to allow CloudFront OAC (console gives the policy snippet).

3) Create DynamoDB table

- Console → DynamoDB → Create table:
  - Table name: `studentData`
  - Partition key: `studentid` (Number or String)
  - Provisioned: On-demand (recommended for simple projects)
- Note the table name/region for Lambda config.

4) Create IAM role for Lambda(s)

Create a role with trust policy for Lambda and attach a policy allowing DynamoDB access and S3 read (if Lambda needs S3). Example policy JSON:

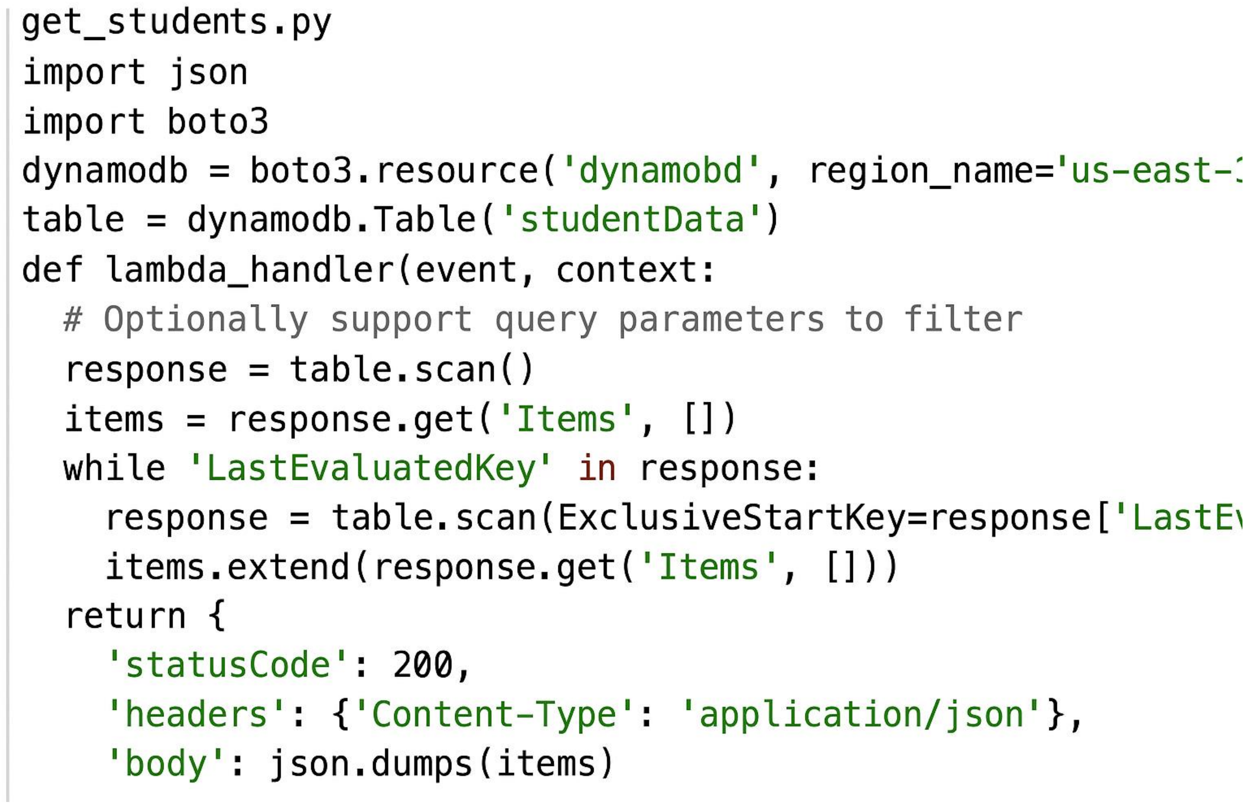


```
"Version: "2012-10-17,
"Statement": {
  {
    "Effect": "Allow",
    "Action":
      "dynamoodd::PutItem,
      "dynamoodb:GetItem,
      "dynamoodb:Query,
      "dynamoodb:Scan,
      "dynamoodb:UpdateItem
    "Resource":
      "arn:aws:dynamoodb:us-east-1:
      123456789012:table/studentData
    }
    "Effect": "Allow",
    "Action":
      "s3:GetObject
    "Resource": my-serverless-stuudent-
    app/*
  }
  "Effect": "Allow",
  "Action":
    "logs:CreateLogGroup
    "logs:CreateLogStream
    "logs:PutLogEvents
  "Resource": arn:aws:logs:us-east-1::
```

5) Create Lambda functions

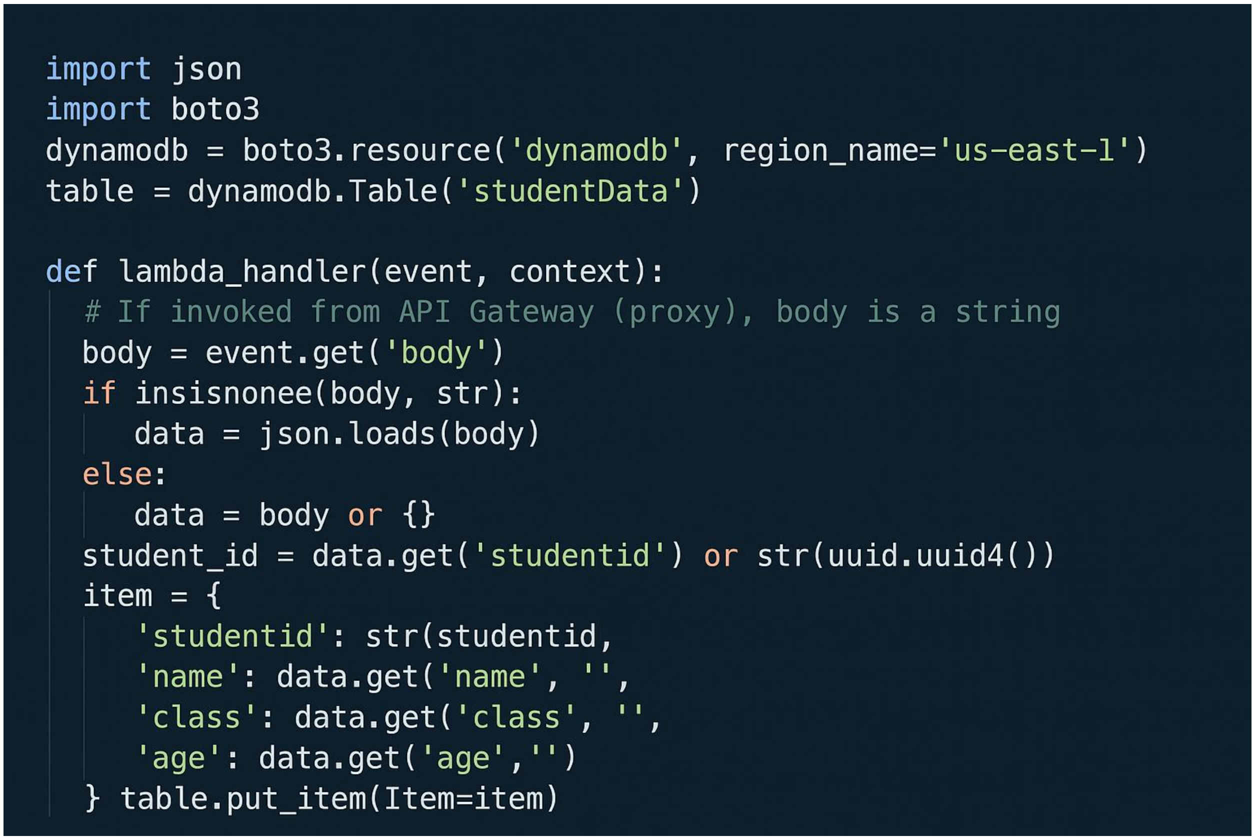
GET lambda (list students) POST lambda (add student)

Python example (remember to set IAM role with DynamoDB read permissions):



```
get_students.py
import json
import boto3
dynamodb = boto3.resource('dynamobd', region_name='us-east-
table = dynamodb.Table('studentData')
def lambda_handler(event, context:
  # Optionally support query parameters to filter
  response = table.scan()
  items = response.get('Items', [])
  while 'LastEvaluatedKey' in response:
    response = table.scan(ExclusiveStartKey=response['LastEv
    items.extend(response.get('Items', []))
  return {
    'statusCode': 200,
    'headers': {'Content-Type': 'application/json'},
    'body': json.dumps(items)
```

POST lambda (add student)



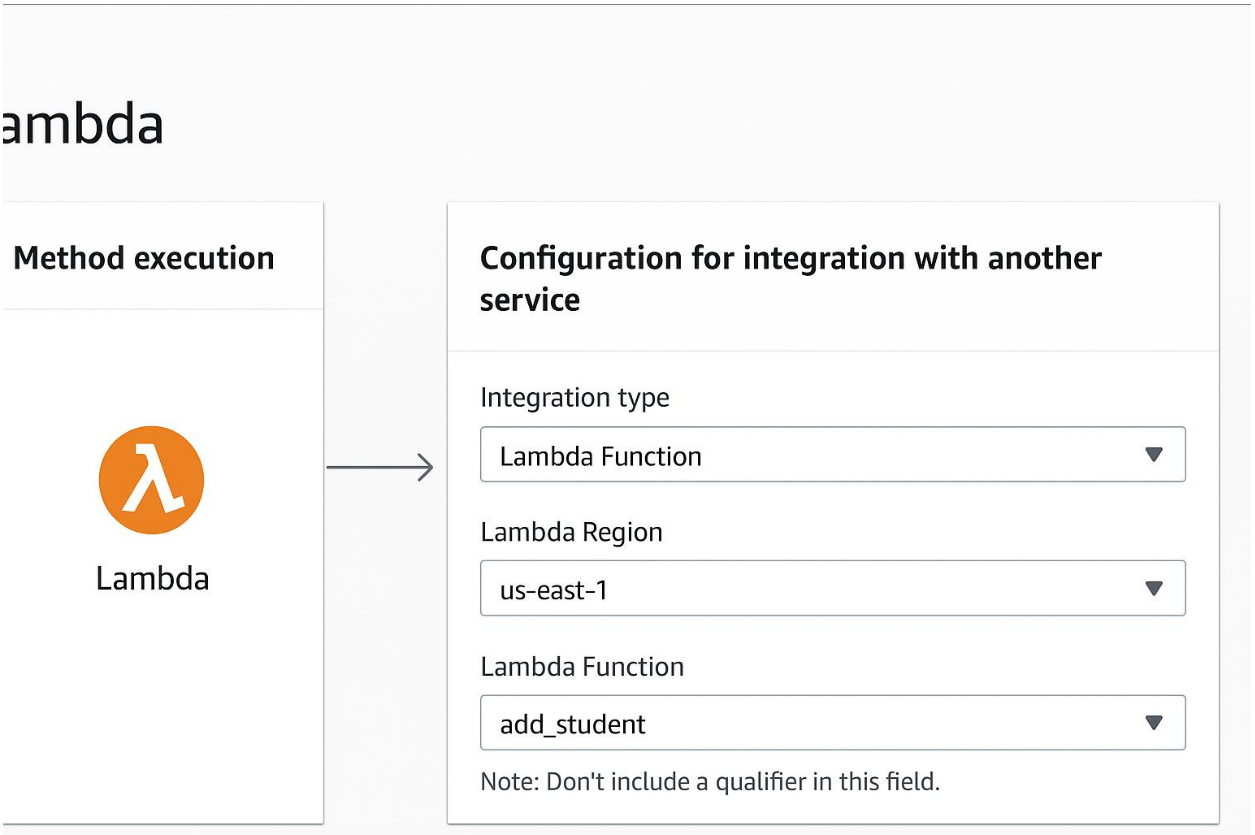
```
import json
import boto3
dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
table = dynamodb.Table('studentData')

def lambda_handler(event, context):
    # If invoked from API Gateway (proxy), body is a string
    body = event.get('body')
    if insisnonee(body, str):
        data = json.loads(body)
    else:
        data = body or {}
    student_id = data.get('studentid') or str(uuid.uuid4())
    item = {
        'studentid': str(studentid,
        'name': data.get('name', '',
        'class': data.get('class', '',
        'age': data.get('age','')
    } table.put_item(Item=item)
```

- Create two Lambda functions (GET and POST), upload code (zip or inline), set handler, runtime (Python 3.9/3.11).
- Configure environment variable `TABLE_NAME=studentData` if you prefer

6) Create API Gateway and integrate Lambdas

1. Console → API Gateway → Create REST API (or HTTP API).
    a. I recommend **HTTP API** for simplicity (lower latency & cost) and it works well with Lambda.
2. Create routes:
    a. `GET /students` → Integration → GET Lambda
    b. `POST /students` → Integration → POST Lambda
3. Enable CORS for routes (allow origin from CloudFront domain or * for testing).
4. Deploy the API (for REST API: stages; for HTTP API: default URL).
5. Copy invoke URL.



ambda

Method execution

Lambda

Configuration for integration with another service

Integration type: Lambda Function

Lambda Region: us-east-1

Lambda Function: add_student

Note: Don't include a qualifier in this field.

7) Update frontend (script.js)

Example fetch usage:

```
const API_ENDPOINT = "https://abcd1234.execute-api.us-east-1.amazonaws.com"; // your API

async function getStudents(){
  const res = await fetch(`${API_ENDPOINT}/students`);
  const data = await res.json();
  console.log(data);
}

async function addStudent(payload){
  const res = await fetch(`${API_ENDPOINT}/students`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(payload)
  });
  return await res.json();
}
```

- Replace API_ENDPOINT with actual invoke URL.
- Upload updated `script.js` to S3 and invalidate CloudFront cache (optional).

8) Enable CORS and Security

- In API Gateway enable CORS (allow CloudFront domain).
- Use Amazon Cognito for auth:
  - Create Cognito User Pool → App Client → configure JWT authorizer in API Gateway.
  - Update frontend to sign-in and pass `Authorization: Bearer <id_token>` header.

9) Test end-to-end

1. Open CloudFront URL (serves `index.html`).
2. Use UI to POST a new student and GET the list.
3. Verify DynamoDB table has item.
4. Check CloudWatch Logs for Lambda invocations.

10) Monitoring & Logging

- CloudWatch Logs: verify logs for both Lambdas.
- CloudWatch Alarms:
  - Lambda errors > threshold
  - API 5xx errors > threshold