**Serverless Web Application**

**(Employee Information Portal)**

**Name:** Sridivya Puttoju

**Student id:** 700767142

**About Project:**

This project is a simple web-based Employee Information Portal. It allows users to enter, save, and view employee details like ID, name, and age. The interface is clean, user-friendly, and easy to navigate. Employee profiles are displayed in a structured table format. The system supports real-time data entry and retrieval through form interactions.

This project implements a web-based system using Amazon Web Services (AWS) to manage both static content (such as web pages and assets) and dynamic data interactions. It leverages multiple AWS services to process user requests, store employee information, and ensure fast, secure delivery of content across the platform.

**AWS Services Used:**

**Amazon CloudFront –** Fast content delivery system (CDN) for global access

**Amazon S3 –** Stores and serves static files like HTML, CSS, JS, and images

**Amazon API Gateway –** Handles user requests and connects them to Lambda functions

**AWS Lambda –** Executes backend logic for saving and retrieving employee data without server management

**Amazon DynamoDB –** NoSQL database used to store and manage employee profiles

**AWS IAM –** Manages secure access and permissions between services

**Services Work Together:**

**User Request:** The user interacts with the portal (e.g., to save or view employee details).

**CloudFront:** Amazon CloudFront delivers the static web interface quickly by caching it near the user's location**.**

**S3:** Static website files (HTML, CSS, JS) are stored in Amazon S3 and served through CloudFront.

**API Gateway:** For dynamic actions like saving or retrieving employee data, requests are routed to API Gateway.

**Lambda:** AWS Lambda runs the backend logic (e.g., storing a new employee record or fetching employee data).

**DynamoDB:** Amazon DynamoDB stores employee profiles and provides fast access when queried.

**Step by Step workflow:**

**User Request:**

* The user accesses the application through a browser.

**CloudFront:**

* Amazon CloudFront quickly serves the HTML, CSS, and JS files from its edge locations.
* These static files are stored in an Amazon S3 bucket.

**User Submits a Form (e.g., Save/View Employee Info):**

* The browser sends a request (GET or POST) to the backend.

**API Gateway:**

* The request is routed through Amazon API Gateway.
* It determines which backend function (Lambda) to trigger.

**Lambda Function:**

* AWS Lambda runs the appropriate function.
* For "Save": it parses the form data and prepares it for storage.
* For "View": it prepares a query to fetch data.

**DynamoDB:**

* Lambda connects to DynamoDB.
* It either stores new employee data or retrieves existing records.

**Response:**

* Lambda returns the result to API Gateway.
* API Gateway sends the response to the user’s browser.
* The frontend updates dynamically with success messages or employee data.

**Procedure:**

**DynamoDB:**

* Navigate to the DynamoDB console in your AWS account.
* Click on ‘Create table’ to begin setting up a new table.
* Provide a Table Name and define the Primary Key (e.g., empId as the partition key).
* Click ‘Create’ to finalize the setup - table will be ready for storing data.

**S3:**

* Open the Amazon S3 console and click ‘Create bucket’.
* Enter a unique bucket name, select your preferred region, and click Create.
* After the bucket is created, navigate to the Permissions tab to adjust access settings, such as Bucket Policy or IAM roles.
* In the Objects tab, click Upload and drag your website files (e.g., profile.html, script.js) into the upload window.
* Go to the Properties tab, scroll down to Static website hosting, and enable it.
* Copy the endpoint URL provided and open it in browser to view your live static website.

**Profile.html:**

<html>

<head>

<title>Profile</title>

<style>

body {

background-image: url('https://d3ncv2il1vdtm.cloudfront.net/image.jpg');

background-size: cover;

background-repeat: no-repeat;

background-attachment: fixed;

font-family: Arial, sans-serif;

color: #000;

}

input[type="text"] {

background-color: lightgray;

padding: 5px;

border: 1px solid #ccc;

border-radius: 4px;

}

input[type="submit"] {

margin-top: 10px;

padding: 8px 16px;

border: none;

border-radius: 5px;

background-color: #4CAF50;

color: white;

cursor: pointer;

}

input[type="submit"]:hover {

background-color: #45a049;

}

table, th, td {

border: 1px solid black;

border-collapse: collapse;

background-color: white;

}

th, td {

padding: 8px;

}

</style>

</head>

<body>

<div align="center">

<h1>Employee Information Portal</h1>

Employee ID:

<br>

<input type="text" name="employeeid" id="empid" style="background-color: lightgray;">

<br> First name:

<br>

<input type="text" name="firstname" id="fname" style="background-color: lightgray;">

<br> Last name:

<br>

<input type="text" name="lastname" id="lname" style="background-color: lightgray;">

<br> Employee Age:

<br>

<input type="text" name="empage" id="empage" style="background-color: lightgray;">

<br><br>

<input type="submit" id="saveprofile" value="Save Profile">

<br>

<p id="profileSaved"></p>

<br>

<input type="submit" id="getprofile" value="View all the Employee Profiles">

<br><br>

<div id="showProfile">

<table id="employeeProfile">

<colgroup>

<col style="width:20%">

<col style="width:20%">

<col style="width:20%">

<col style="width:20%">

</colgroup>

<tbody>

<tr>

<th>Employee ID</th>

<th>First name</th>

<th>Last Name</th>

<th>Employee Age</th>

</tr>

</tbody>

</table>

</div>

</div>

<script src="script.js"></script>

<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.0/jquery.min.js"></script>

</body>

</html>

**Script.js:**

//add your api here below

var API\_ENDPOINT = "https://uqw334q47f.execute-api.us-east-2.amazonaws.com/prod"

//AJAX GET REQUEST

document.getElementById("saveprofile").onclick = function(){

var inputData = {

"empId":$('#empid').val(),

"empFirstName":$('#fname').val(),

"empLastName":$('#lname').val(),

"empAge":$('#empage').val()

};

$.ajax({

url: API\_ENDPOINT,

type: 'POST',

data: JSON.stringify(inputData) ,

contentType: 'application/json; charset=utf-8',

success: function (response) {

document.getElementById("profileSaved").innerHTML = "Profile Saved!";

},

error: function () {

alert("error");

}

});

}

//AJAX GET REQUEST

document.getElementById("getprofile").onclick = function(){

$.ajax({

url: API\_ENDPOINT,

type: 'GET',

contentType: 'application/json; charset=utf-8',

success: function (response) {

$('#employeeProfile tr').slice(1).remove();

jQuery.each(response, function(i,data) {

$("#employeeProfile").append("<tr> \

<td>" + data['empId'] + "</td> \

<td>" + data['empFirstName'] + "</td> \

<td>" + data['empLastName'] + "</td> \

<td>" + data['empAge'] + "</td> \

</tr>");

});

},

error: function () {

alert("error");

}

});

}

**CloudFront:**

* Open the Amazon CloudFront console and click “Create Distribution”.
* Under Origin settings, choose S3 bucket as the origin.
* Set the protocol policy to redirect HTTP to HTTPS for secure access.
* Configure additional settings such as Viewer Protocol Policy, Caching behavior, and SSL certificate (if using a custom domain).
* Click Create Distribution and wait for the deployment to complete.
* After creating the distribution, update your S3 bucket policy to allow CloudFront access.
* After that, test the CloudFront domain (or custom domain) to confirm successful website delivery.

**Lambda:**

* Go to the AWS Lambda Console and click “Create function”.
* Choose Author from scratch, give your function a name like ‘getEmployee’, and select Python 3.9 as the runtime.
* Assign or create an IAM role with permissions to access DynamoDB and click create function.
* Write the Lambda function code in Python to handle saving and retrieving employee details to/from DynamoDB.
* Create a Test Event simulating a POST request for saving data and a GET request for retrieving employee profiles.
* After verifying the function works correctly, deploy it and link it to an API Gateway to handle HTTP requests from your frontend.
* Repeat the same steps for the ‘insertEmployee’ function as well.

**getEmployee:**

import json

import boto3

def lambda\_handler(event, context):

dynamodb = boto3.resource('dynamodb', region\_name='eu-central-1')

table = dynamodb.Table('employeeProfile')

response = table.scan()

data = response['Items']

while 'LastEvaluatedKey' in response:

response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])

data.extend(response['Items'])

return data

**insertEmployee:**

import json

import boto3

# create a DynamoDB object using the AWS SDK

dynamodb = boto3.resource('dynamodb')

# use the DynamoDB object to select our table

table = dynamodb.Table('employeeProfile')

# define the handler function that the Lambda service will use as an entry point

def lambda\_handler(event, context):

# extract values from the event object we got from the Lambda service and store in a variable

firstname = event['empFirstName']

id=event['empId']

lastname=event['empLastName']

age=event['empAge']

# write name and time to the DynamoDB table using the object we instantiated and save response in a variable

response = table.put\_item(

Item={

'empId': id,

'empAge':age,

'empFirstName':firstname,

'empLastName':lastname

})

# return a properly formatted JSON object

return {

'statusCode': 200,

'body': json.dumps('Hello from Lambda, ' + firstname)

}

**API Gateway:**

* Open the API Gateway Console and click ‘Create API’. Select REST API and click ‘Build’.
* Give an API name and change API endpoint type to ‘Edge-optimized’ and click create API.
* After creating, go to resources choose ‘Create Resource’ to define a new API endpoint path.
* Under the created resource, click ‘Create Method’ (e.g., GET, POST) to specify how the frontend communicates with the backend.
* For integration type, select ‘Lambda Function’, and enter the name of the function you want to link.
* Grant API Gateway permission to invoke the selected Lambda function when prompted.
* Go to Actions → Deploy API, choose a deployment stage, and click Deploy.
* After deployment, we will receive an Invoke URL that your frontend can use to send HTTP requests.

**Benefits of using AWS for this Project:**

* **Scalability**: Easily handles growing data and user requests without performance issues.
* **Cost-Efficiency**: Pay only for what you use — no need to maintain physical servers.
* **High Availability**: AWS services like S3, Lambda, and DynamoDB ensure application is always accessible.
* **Security**: Built-in security features like IAM roles, encryption, and API Gateway throttling protect your data.
* **Speed & Performance**: CloudFront accelerates content delivery, while Lambda and DynamoDB respond in milliseconds.
* **Serverless Architecture**: AWS Lambda removes the need for backend server management, reducing complexity.
* **Simple Integration**: AWS services work seamlessly together — from hosting (S3) to APIs (API Gateway) to storage (DynamoDB).

**Conclusion:**

By combining S3 and CloudFront for static delivery, API Gateway and Lambda for business logic, and DynamoDB for database operations, this AWS-based setup ensures high performance, flexibility, and minimal maintenance overhead.

**References:**

* 1. CloudFront [https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction .html](https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction%20.html)
  2. S3 <https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html>
  3. API Gateway

<https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>

* 1. Lambda

<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>

* 1. DynamoDB <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>