

[Type here]

## **SERVERLESS WEBSITE HOSTING USING AWS S3, CLOUDFRONT, AWS LAMBDA, DYNAMODB, API GATEWAY**

Lets explore serverless application hosting. We will be hosting our application in S3 with CloudFront as the frontend, and our backend will consist of API Gateway, Lambda, and DynamoDB.

🚦 First of all, why serverless?

If an app would go down for various reasons like memory issues, scaling problems, server maintenance/downstream, or even upstream issues. These all problems occur cause of servers. Wherever there are servers, you have to manage them. So why don't we host our apps in a serverless architecture? It's cost-effective, requires less maintenance, and offers greater flexibility.

You must be familiar with serverless computing, which has become an integral part of the market. The deployment of serverless architecture reduces manual engagement by allowing developers to focus on their core product rather than managing and operating servers or runtimes, whether in the cloud or on-premise. This reduces the burden, improves scalability, and enhances overall productivity.

Serverless architecture is gaining traction in the market, with the serverless architecture market size exceeding \$7 billion in 2020 and expected to grow by over 20% annually from 2021 to 2027.

🚦 Is Serverless Truly Serverless?

Serverless architecture does not mean operating without servers. Instead, it refers to the hassle-free management of servers by the cloud provider. Your applications and workloads still run on servers, but the cloud provider takes care of server handling, eliminating the need for provisioning, scaling, and maintenance.

🚦 Here's a high-level overview of our project:

We will create a dynamic web application with features for inserting and retrieving employee data.

The web application will be hosted in a serverless architecture.

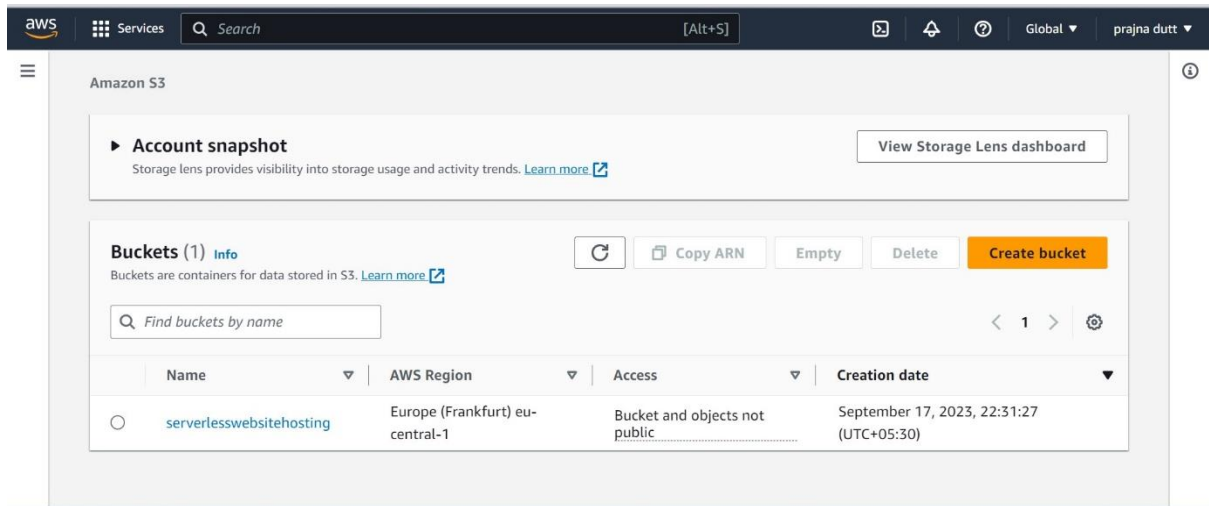
CloudFront will serve as the frontend, directing requests to the appropriate AWS services.

We'll build the backend with Lambda functions for processing API requests and DynamoDB for data storage.

The main idea here is that our entire project, including the web application and backend, will be hosted in a serverless environment. This means we won't have to worry about managing servers, scaling, or infrastructure maintenance. Everything will scale automatically based on demand.

✓ Hosting a Static Website:

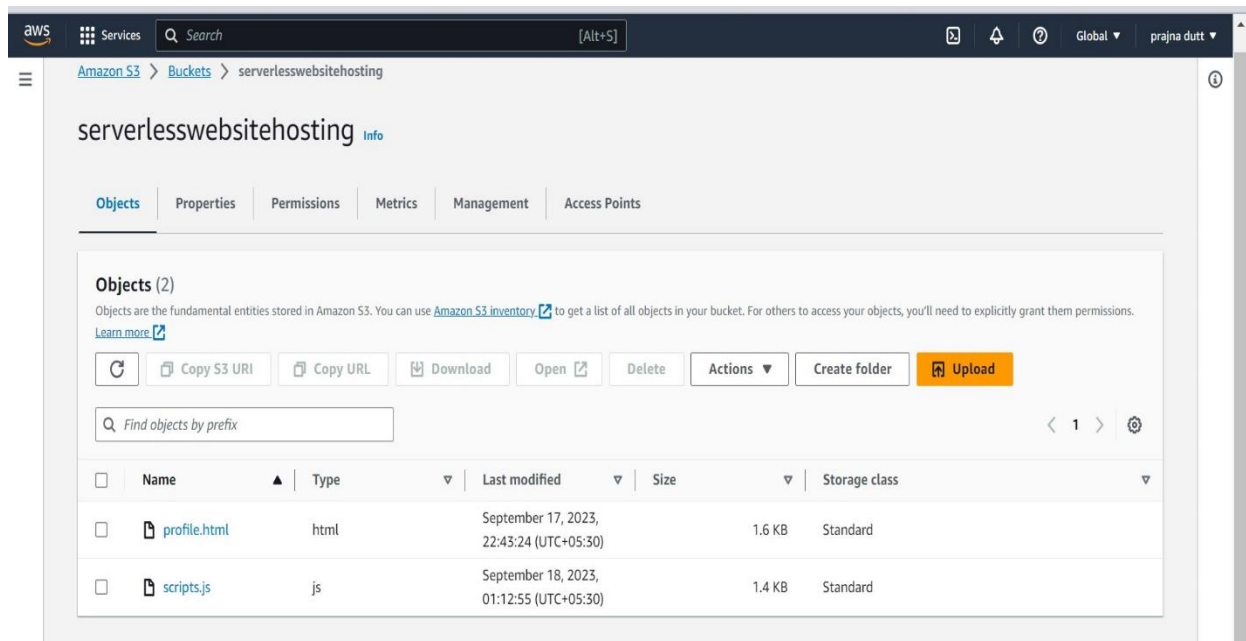
We will focus on hosting a static website in an S3 bucket and accessing it via CloudFront distribution. We'll follow these steps:



Hosting the Website in S3:

1. Create an S3 bucket with a unique name in your preferred AWS region.
2. Configure the bucket to allow public access.
3. Enable static website hosting for the bucket.

Uploading Website Files:



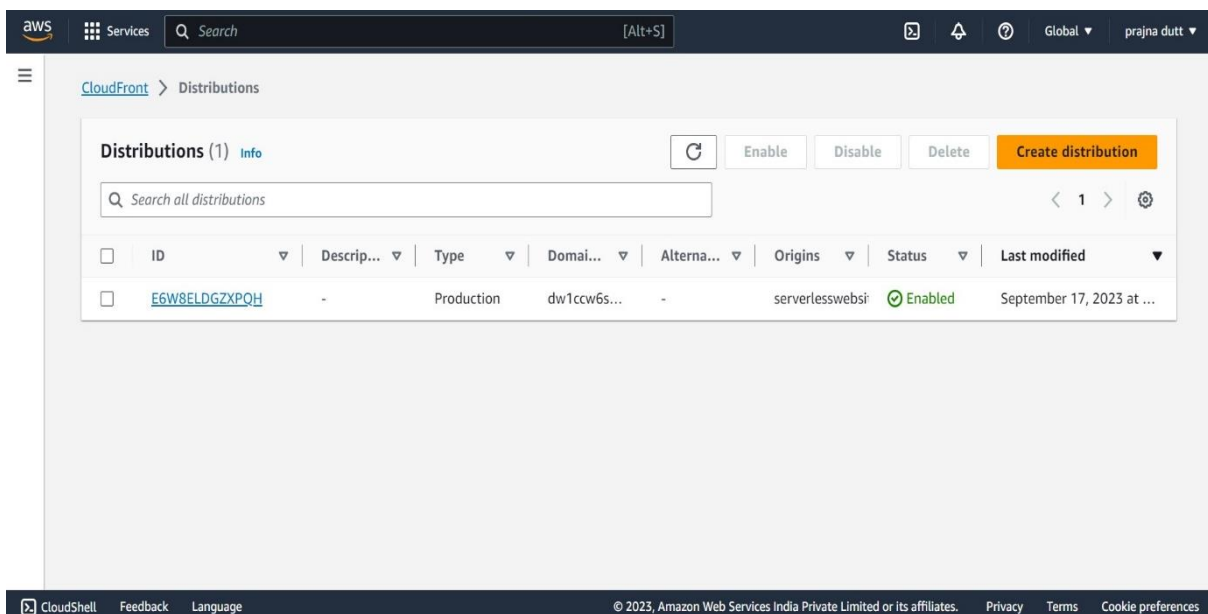
4. Upload the website files, such as profile.html and script.js, to the S3 bucket.

[Type here]

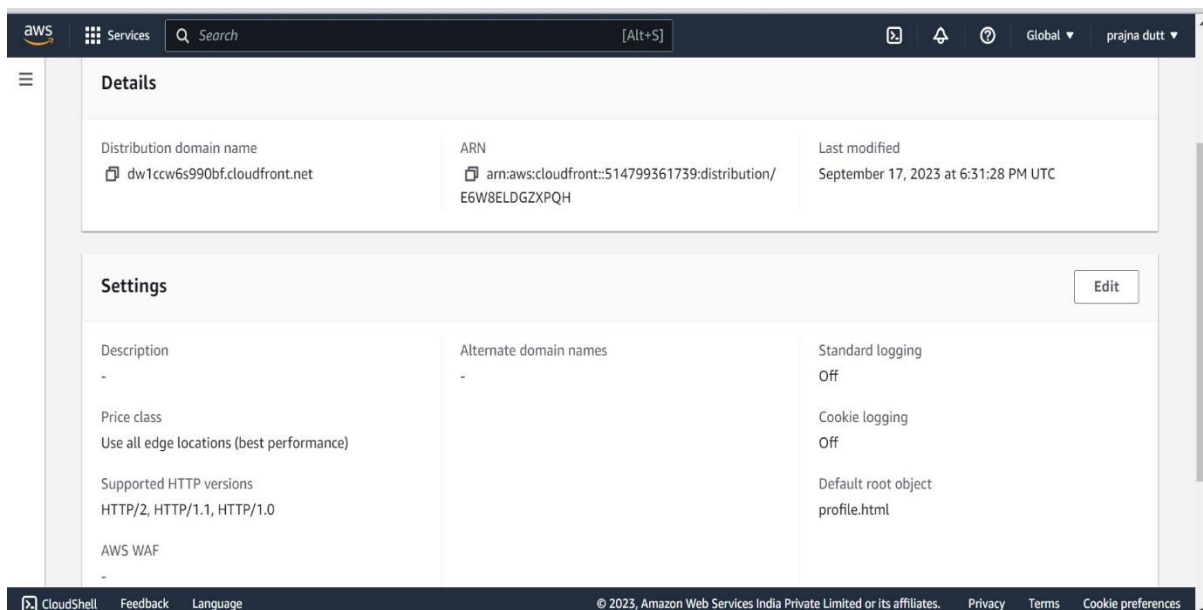
### ✓ Configuring CloudFront:

5. Create a CloudFront distribution and associate it with the S3 bucket.

Set up the default root object in CloudFront to redirect users to profile.html when accessing the root URL.



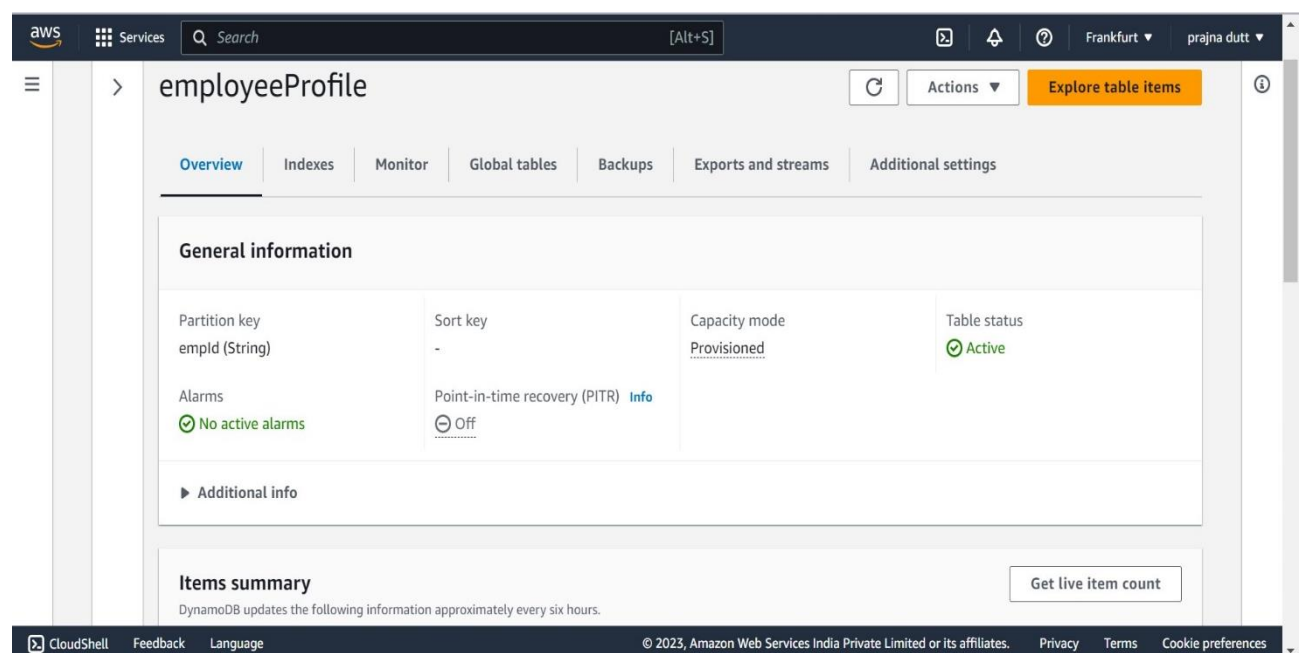
After completing these steps, your website will be accessible via the CloudFront distribution URL, providing a secure and scalable hosting solution.



In this part, we will create our backend. Our backend consists of a serverless DynamoDB database for data storage. To query the database, we will create two Lambda functions: one for inserting data into the DynamoDB table and another for fetching data from the DynamoDB table.

we will be creating two Lambda functions: one to insert data into DynamoDB and one to retrieve data from the DynamoDB table.

✓ Create DynamoDB Table:



I am inside my DynamoDB console, selecting the Frankfurt region for this.

In the DynamoDB console, we have a "Create Table" option.

We will click on this one and provide a table name. For this, I will name the table "EmployeeProfile."

Then, we have to specify a partition key, which is the primary key in DynamoDB. We can also specify a sort key.

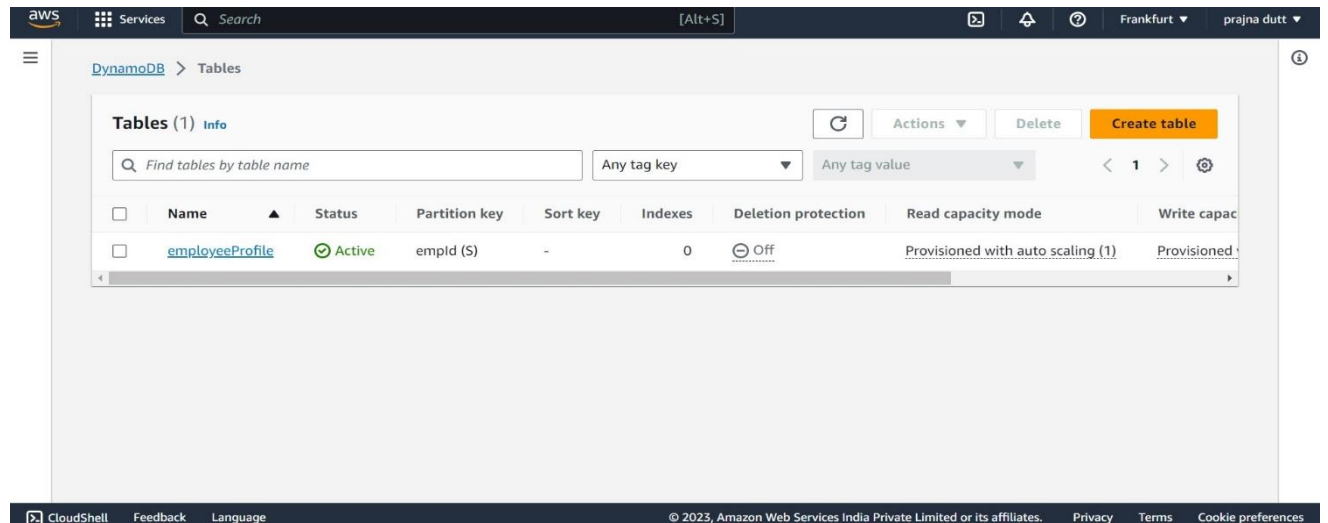
The partition key and sort key together make a unique key in DynamoDB. For the partition key, we will use "EmployeeID."

We'll check our script. If you remember from the first part, this script will attempt to enter data into our DynamoDB table. It will call our API Gateway, which is not yet set up. This script will call the Lambda functions, and the API Gateway will invoke these Lambda functions to insert data into the DynamoDB table.

We'll keep the column name as it is in our script to avoid confusion. So, the partition key is "EmployeeID." After that, I won't change anything else; everything will remain as the default.

[Type here]

I will give a tag; we can set the tag as "Owner" with the value as "MyChannelName" and create the table. DynamoDB table creation will take some time.

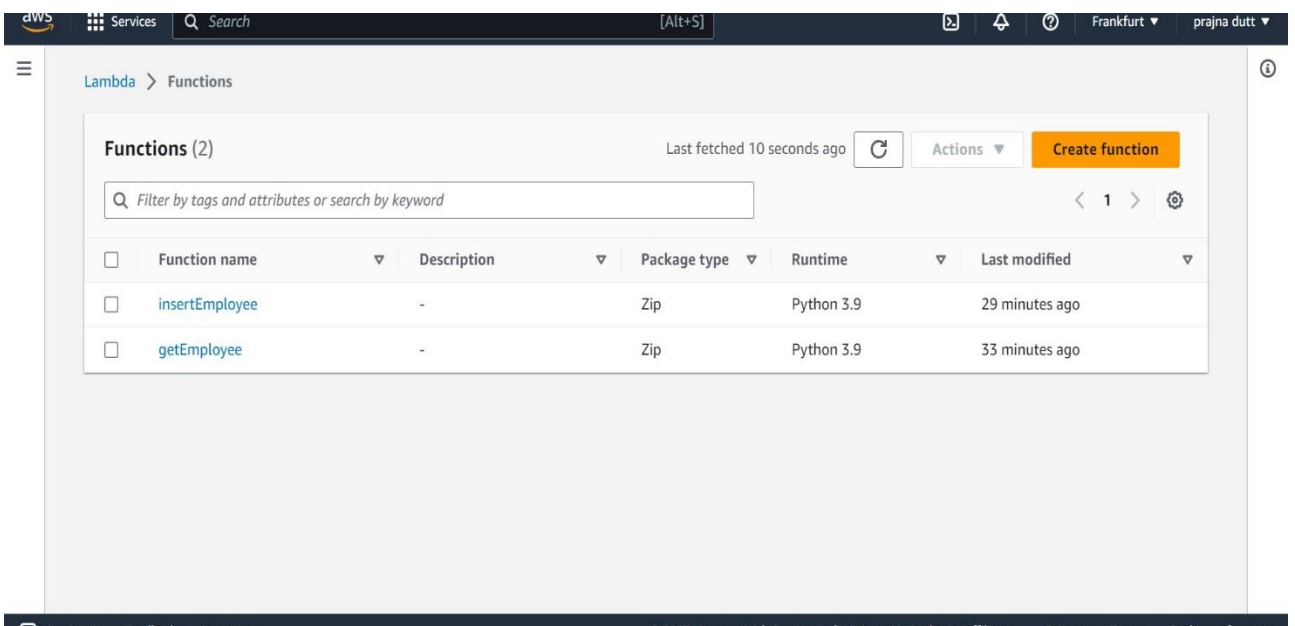


And DynamoDB table will be created.

#### ✓ Create Lambda Functions:

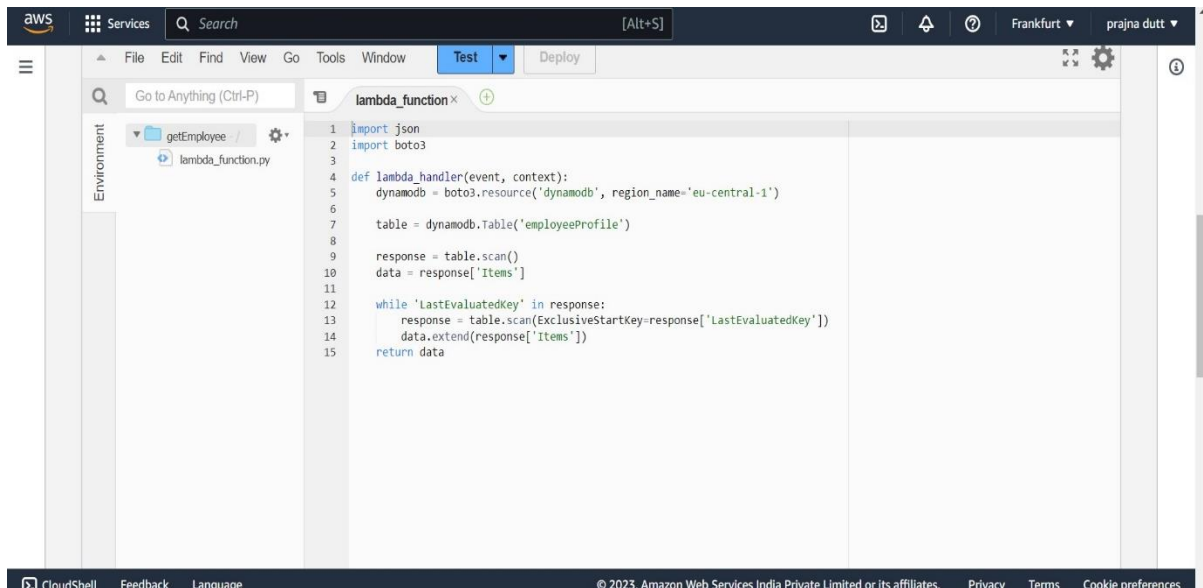
Let's go to our Lambda console and create two Lambda functions. Here's my Lambda console, and I'm using the Frankfurt region. I will create the first function, "GetEmployee," to query employee data from our database.

I will select Python and, besides that, I will only change the execution role. I've already created a role called "LambdaRole," but you can create your own. Just make sure it has DynamoDB access permissions for read and write. I've selected this role and will create the function. Right now, it's an empty function. Let me paste some code here for the "GetEmployee" function.



## Insert Data into DynamoDB:

For inserting data into DynamoDB, we need another Lambda function. If you remember the overview, we need another function for this purpose. So, let's create it. Again, go to "Create

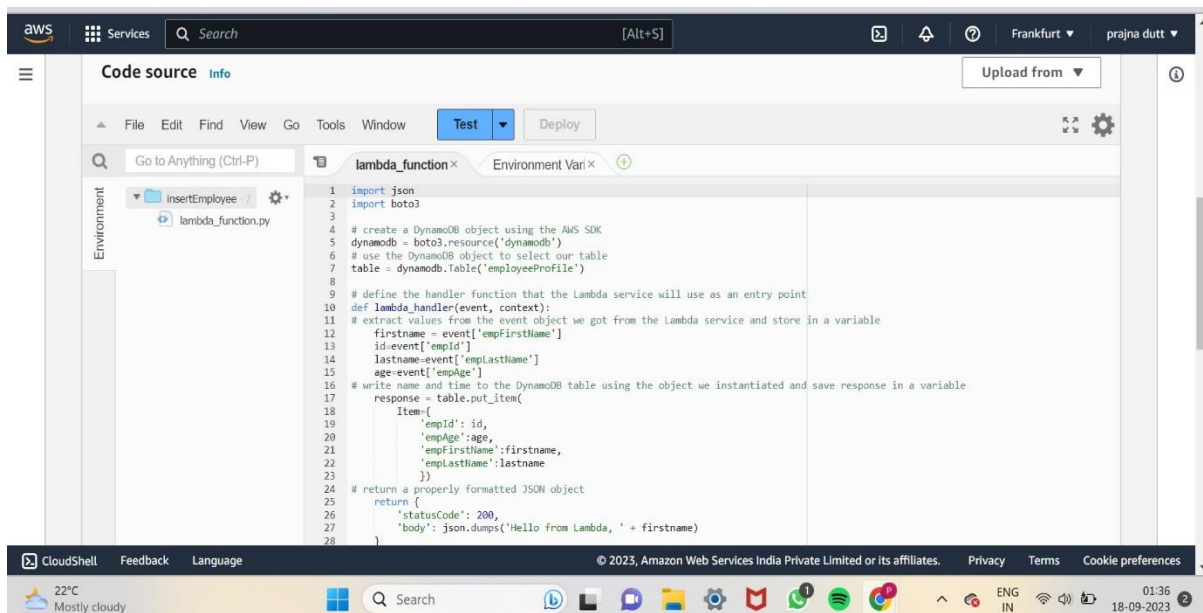


The screenshot shows the AWS Lambda console interface. The 'Code' tab is selected, displaying the Python code for a Lambda function named 'lambda\_function'. The code imports 'json' and 'boto3', then defines a 'lambda\_handler' function. This function creates a DynamoDB resource, scans a table named 'employeeProfile', and returns the data. The environment is set to 'Frankfurt' and the user is 'prajna dutt'.

```
1 import json
2 import boto3
3
4 def lambda_handler(event, context):
5     dynamodb = boto3.resource('dynamodb', region_name='eu-central-1')
6
7     table = dynamodb.Table('employeeProfile')
8
9     response = table.scan()
10    data = response['Items']
11
12    while 'LastEvaluatedKey' in response:
13        response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
14        data.extend(response['Items'])
15    return data
```

Function" and name it "InsertEmployeeData." I will also select Python 3.9 and choose the "LambdaRole" I created. Ensure your Lambda role has write permissions to DynamoDB when creating this "InsertEmployeeData" function.

Let's add the code for this function:



The screenshot shows the AWS Lambda console interface. The 'Code' tab is selected, displaying the Python code for a Lambda function named 'lambda\_function'. The code imports 'json' and 'boto3', then defines a 'lambda\_handler' function. This function creates a DynamoDB resource, scans a table named 'employeeProfile', and returns the data. The environment is set to 'Frankfurt' and the user is 'prajna dutt'.

```
1 import json
2 import boto3
3
4 # create a DynamoDB object using the AWS SDK
5 dynamodb = boto3.resource('dynamodb')
6 # use the DynamoDB object to select our table
7 table = dynamodb.Table('employeeProfile')
8
9 # define the handler function that the Lambda service will use as an entry point
10 def lambda_handler(event, context):
11     # extract values from the event object we got from the Lambda service and store in a variable
12     firstname = event['empFirstName']
13     id=event['empId']
14     lastname=event['empLastName']
15     age=event['empAge']
16
17     # write name and time to the DynamoDB table using the object we instantiated and save response in a variable
18     response = table.put_item(
19         Item={
20             'empId': id,
21             'empAge':age,
22             'empFirstName':firstname,
23             'empLastName':lastname
24         })
25
26     # return a properly formatted JSON object
27     return {
28         'statusCode': 200,
29         'body': json.dumps('Hello from Lambda, ' + firstname)}
```

This code sets up a DynamoDB resource, creates a DynamoDB table client, and takes data from an event to insert into the DynamoDB table. Once we have all the data, we use the "put\_item" method to insert the data into the table. It's important to specify the primary key.

[Type here]

We have created two Lambda functions: one for retrieving employee data and another for inserting data into DynamoDB. We've also created a DynamoDB table, which is a serverless function.

Now, let's deploy this function. After saving and deploying, we can try inserting some data into the DynamoDB table.

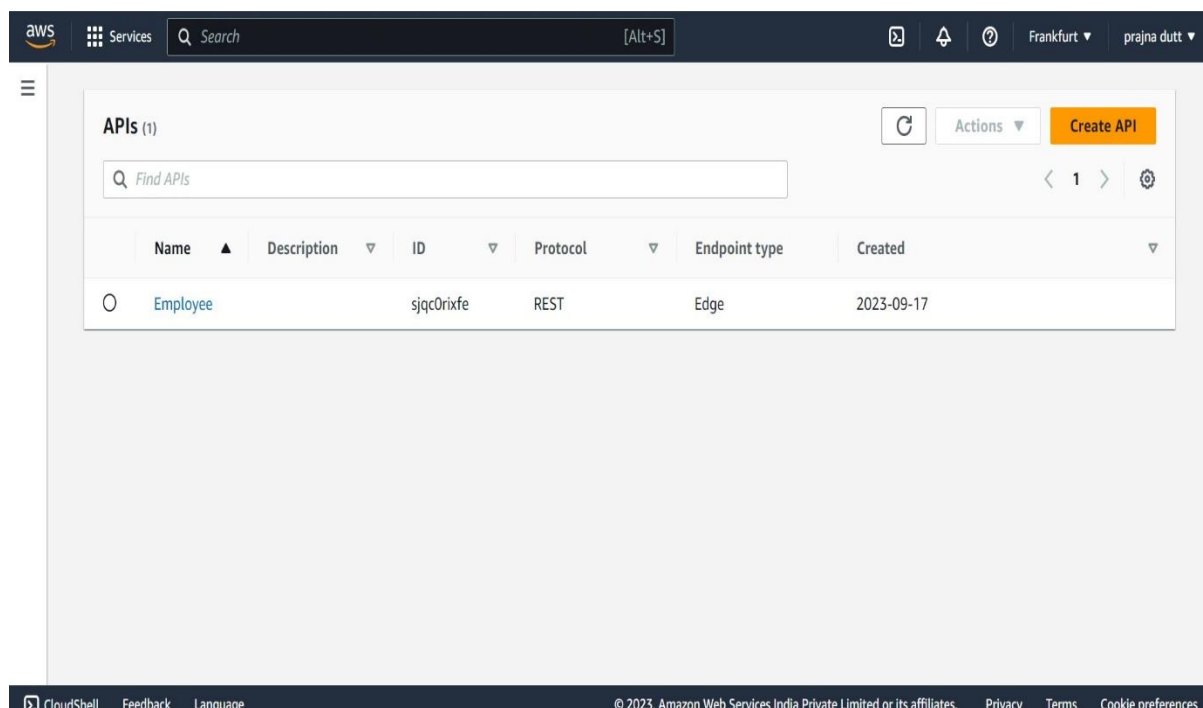
✓ Now we will integrate our frontend and backend using API Gateway.

API Gateway is a service that facilitates API calls to your backend application. As our frontend consists of basic HTML, which needs to communicate with the backend Lambda to insert and retrieve data from DynamoDB, API Gateway plays a crucial role.

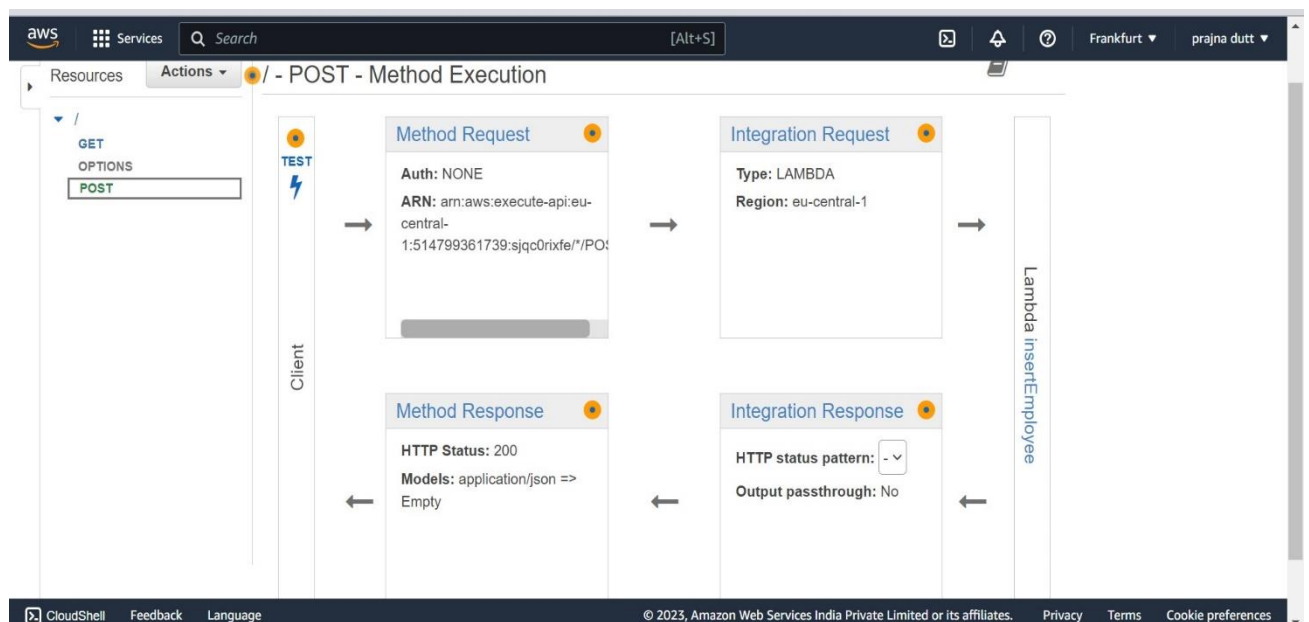
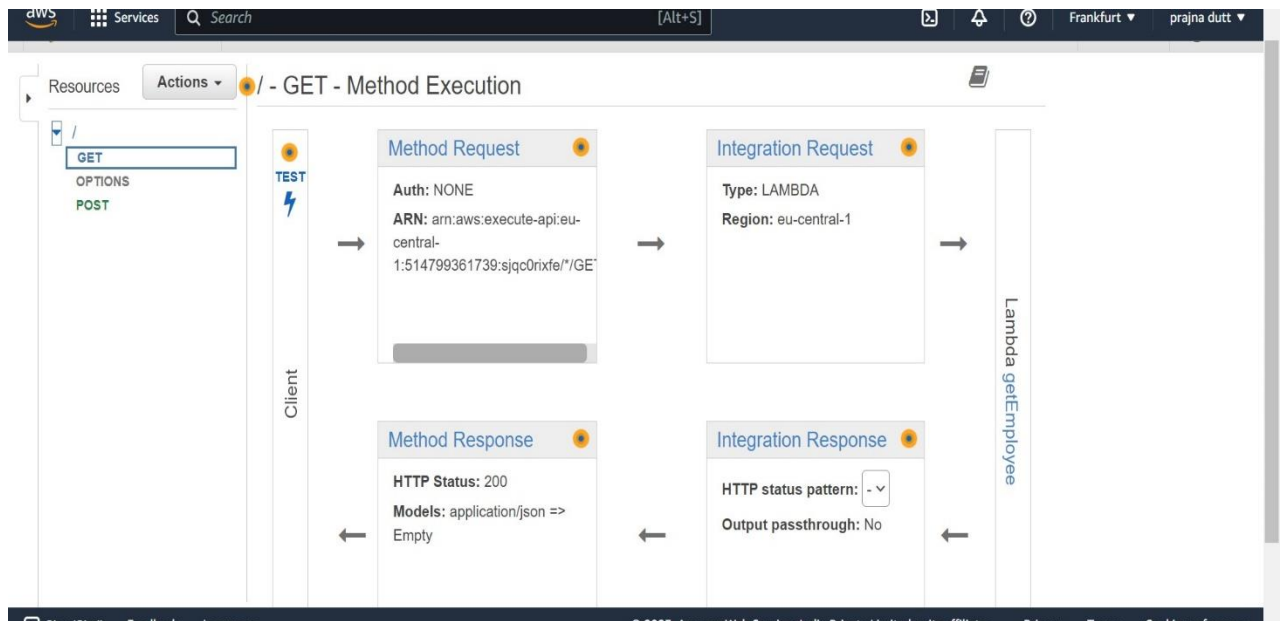
API Gateway, which acts as the bridge between our frontend and backend. API Gateway is an API management tool that sits between clients and a collection of backend services. It serves as the intermediary for requests from the frontend to the backend.

With API Gateway, we will connect these two Lambda functions. Whenever a user requests data from our website, API Gateway will call the appropriate Lambda function to fetch the data. Similarly, when a user inserts data, API Gateway will invoke the Lambda function responsible for saving the data to DynamoDB.

Start by creating our API Gateway. You can find it in the AWS Management Console under "API Gateway." If you don't have one already, create a new API. Choose "REST API" and select "HTTP API" optimized for CloudFront.



Next, create two methods: one for GET requests and one for POST requests. The GET method will call the Lambda function responsible for retrieving data, while the POST method will invoke the Lambda function to insert data into DynamoDB.

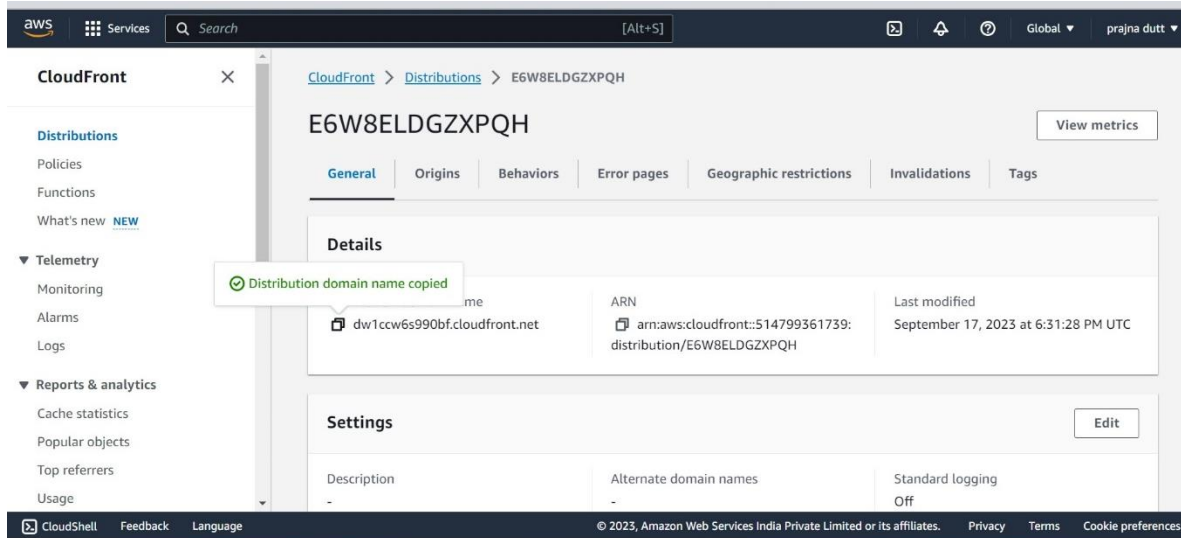


Once you've created these methods, deploy your API to make it live. After deployment, enable CORS (Cross-Origin Resource Sharing) for your API. CORS is essential for allowing your frontend, hosted on a different domain, to communicate with the API Gateway.

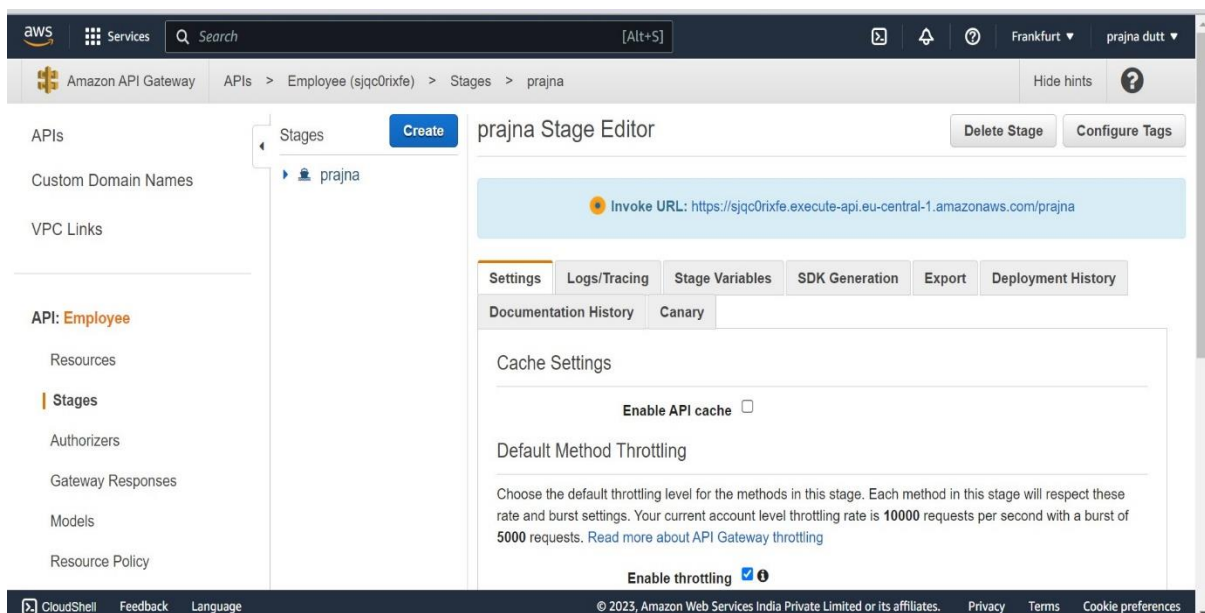


[Type here]

With API Gateway set up, update your frontend code to include the API Gateway URL. If necessary, upload any modified files to your S3 bucket.



Now, your serverless architecture is ready to use. When users interact with your website, the frontend will request data from API Gateway, which, in turn, communicates with Lambda functions to retrieve or insert data into DynamoDB.



This dynamic website is entirely serverless, ensuring scalability, cost-effectiveness, and minimal maintenance. You've successfully built a serverless application that seamlessly integrates frontend and backend components.

Save and View an Employee Profile

Employee ID:

First name:

Last name:

Employee Age:

Employee ID	First name	Last Name	Employee Age
-------------	------------	-----------	--------------

Now we can insert data and fetch the inserted information by clicking View all the Employee Profiles.

## Save and View an Employee Profile

Employee ID:

First name:

Last name:

Employee Age:

Profile Saved!

Employee ID	First name	Last Name	Employee Age
001	Alex	Holth	34
002	abc	xzy	22